Enrico Nardelli
Logic Circuits and Computer Architecture

Appendix A
Digital Logic Circuits

Part 2: Combinational and Sequential Circuits

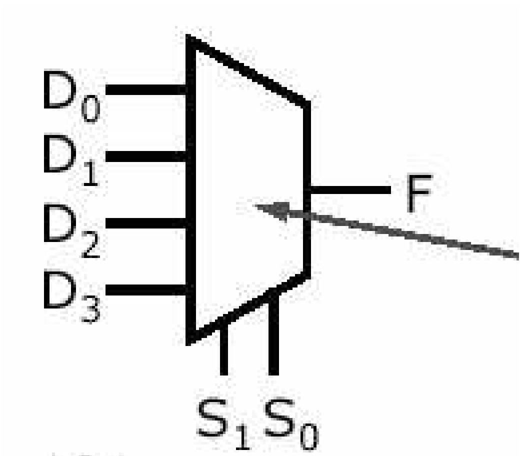
Combinational circuits

- Each of the *m* outputs can be expressed as function of *n* input variables
- Truth table has:
  - *n* input columns
  - *m* output columns
  - 2^n rows (all possible input combinations)

Multiplexer (Mux)

- 2^n data inputs -- 1 output
- *n* controls, to select one of the inputs to be "sent" to the output

Example: 4-to-1 mux

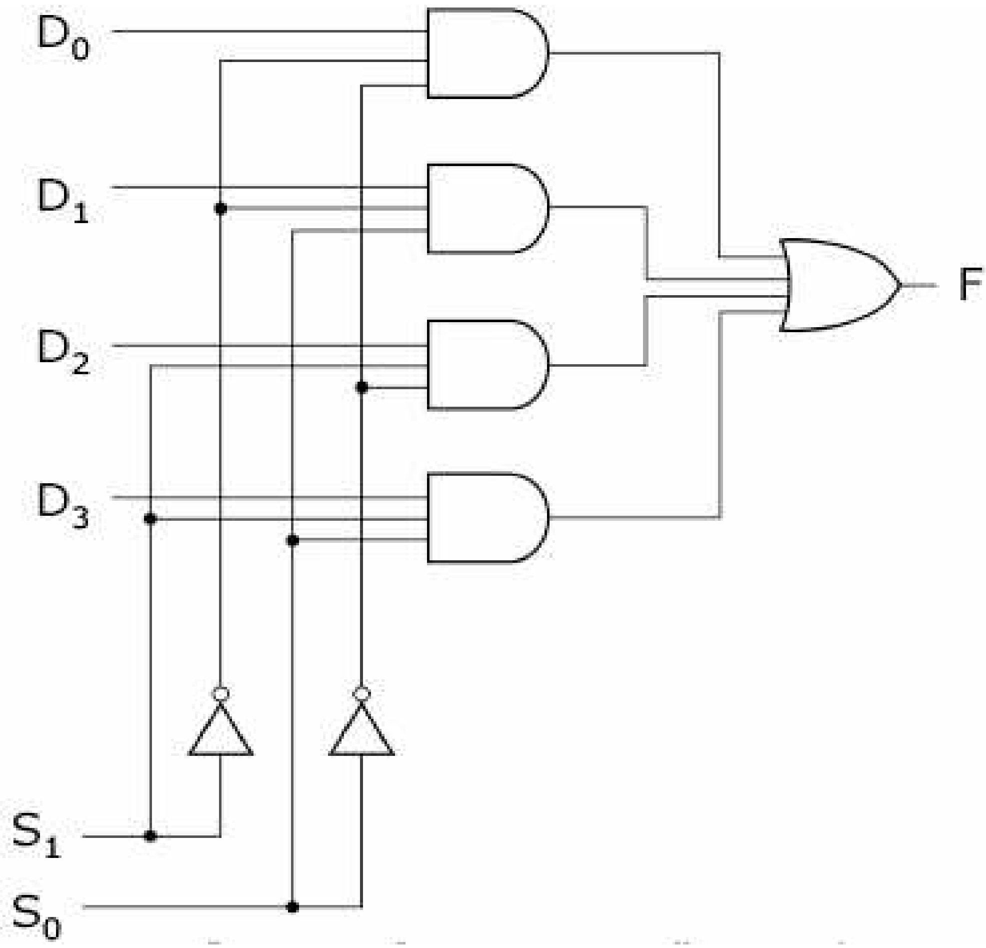


Logic symbol

Truth table

S_1	S_0	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

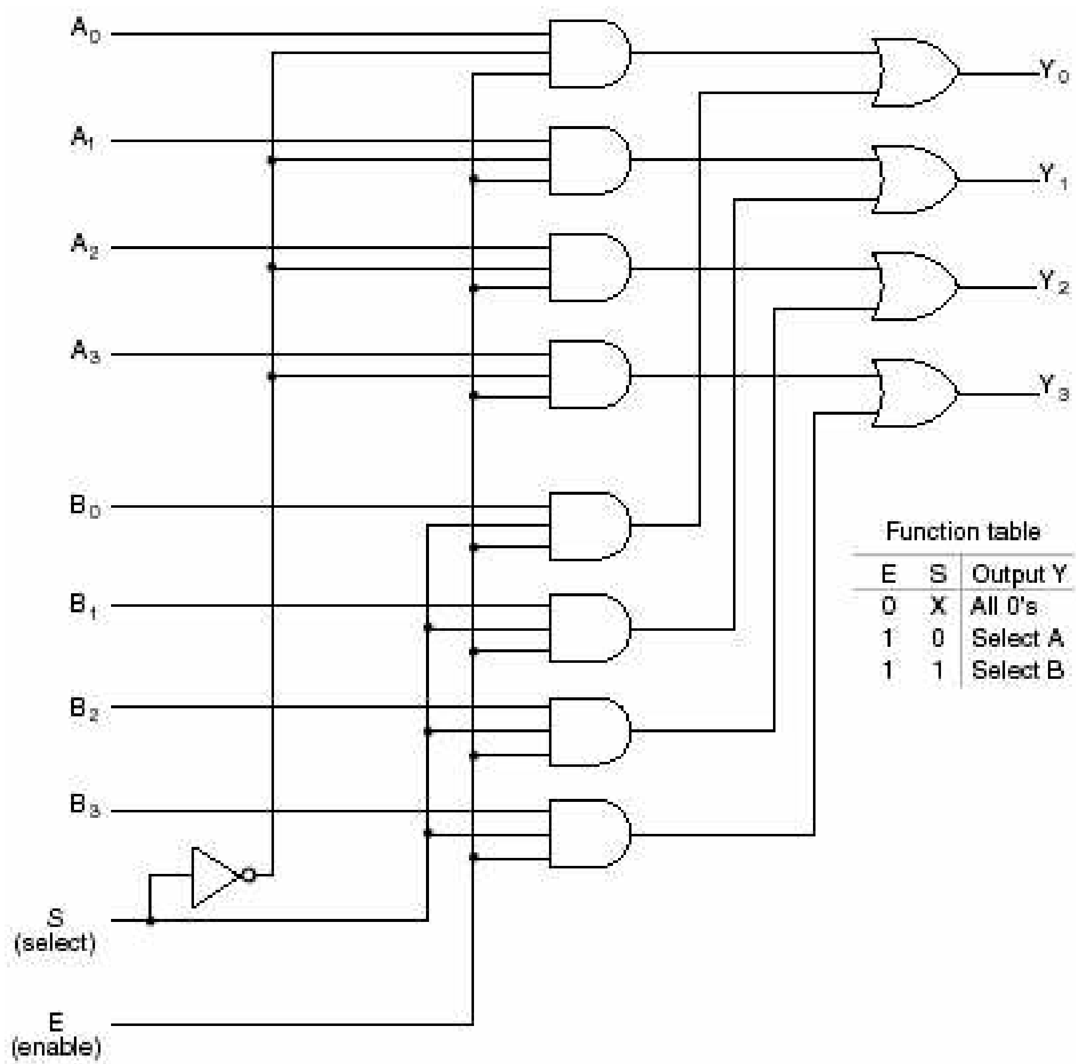
Logic circuit for a 4-to-1 Mux



Exercise

- Consider a 2-to-1 multiplexer:
  - 2 data inputs: D_0 and D_1
  - 1 control input: S_0
  - 1 data output: F
- Write
  - Truth table
  - Logic circuits which implements it
- Extend it to deal with 4 bits at a time

Quadruple 2-to-1 mux



Function table

E	S	Output Y
0	X	All 0's
1	0	Select A
1	1	Select B

De-multiplexer (Demux)

- 1 input -- 2^n data outputs --
- *n* controls, to select exactly one of the outputs to "receive" the input

Example: 1-to-4 demux

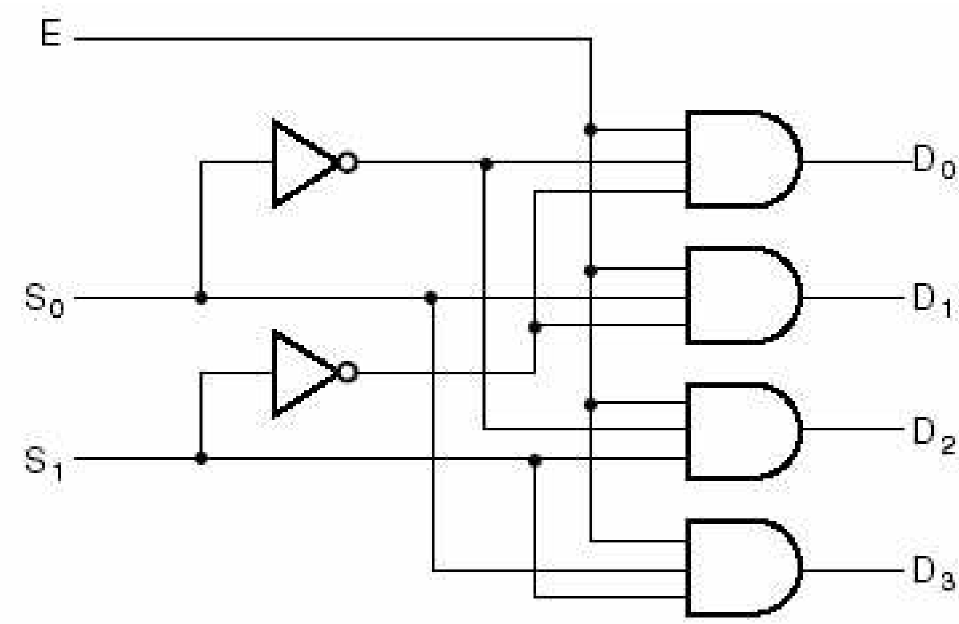
input: E, controls: S_0 , S_1

outputs: D_0 , D_1 , D_2 , D_3

S_0	S_1	D_0	D_1	D_2	D_3
0	0	E			
1	0		E		
0	1			E	
1	1				E

Truth table

Logic circuit for a 1-to-4 Demux

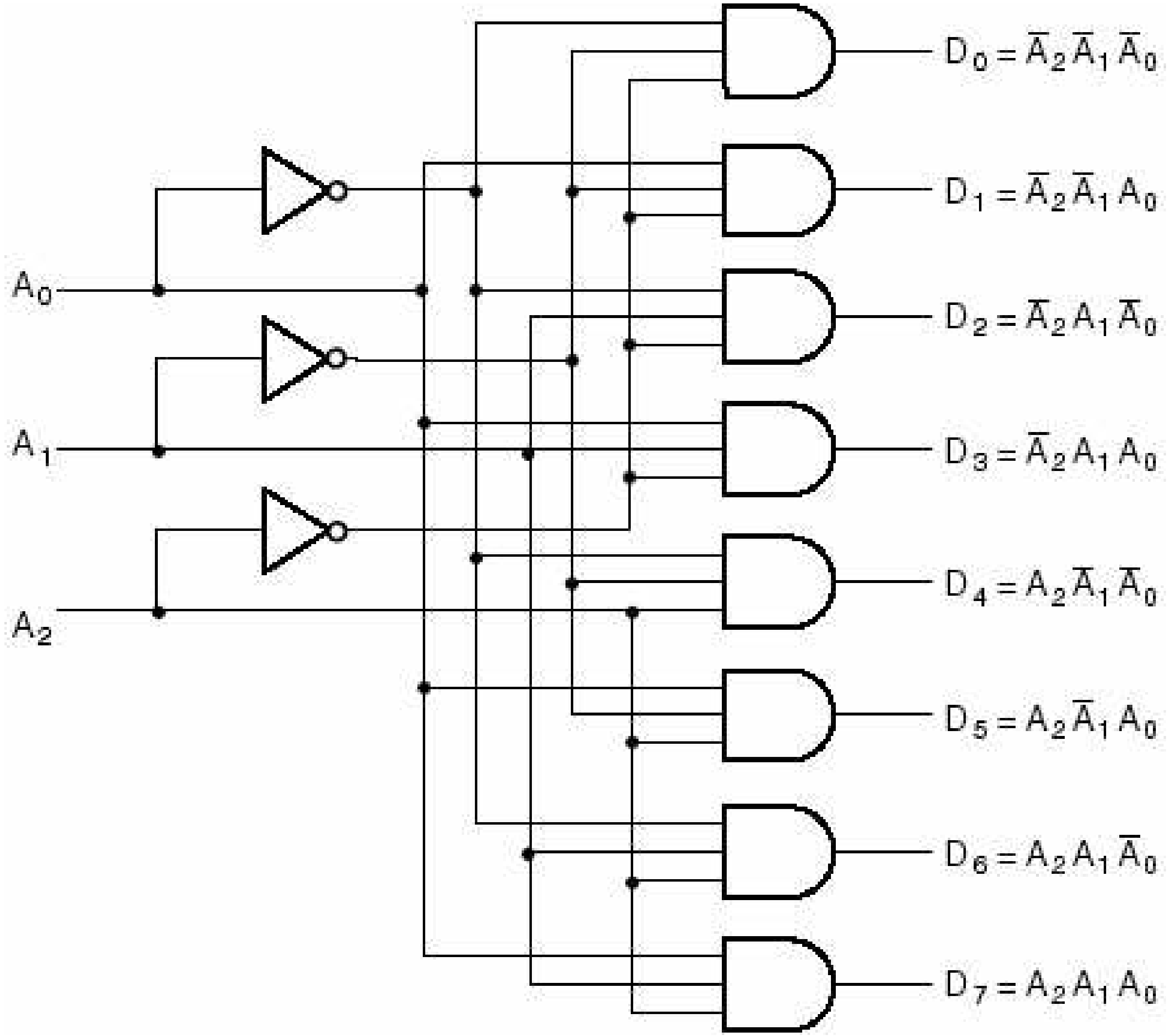


Decoder

- Convert *n* inputs to exactly one of 2^n outputs

  i.e., given an *n*-bit value *i* in input the decoder activates only the *i*-th output line
- Example: a 3-to-8 decoder
  - A 3-bit value in input
  - 8 output lines
  - Write the truth table and the logic circuit

A 3-to-8 decoder

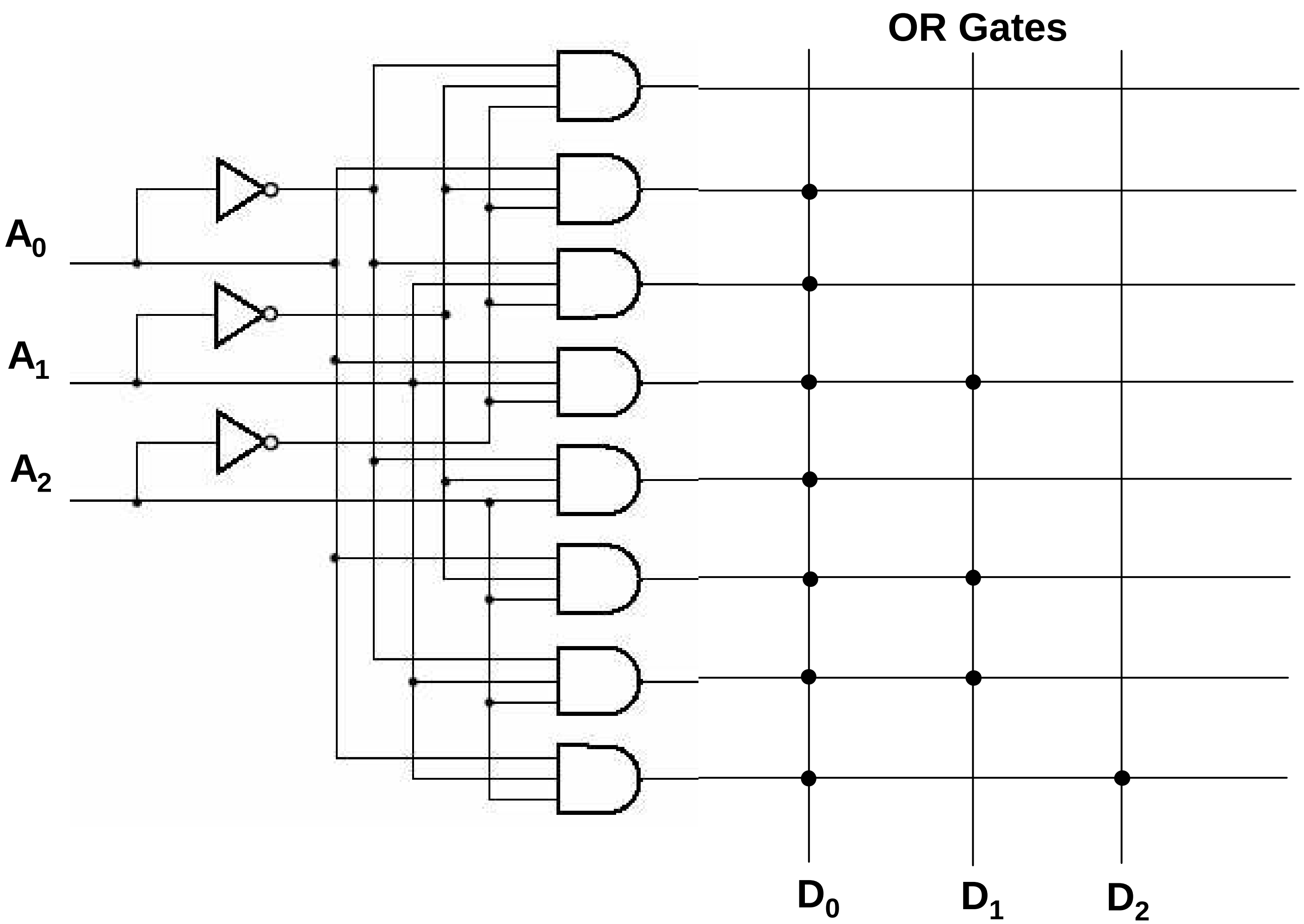


Read Only Memories (ROMs)

- They are just a combinational circuits !
- A simple example for a 8-cell ROM with 3 bits per cell

A_0	A_1	A_2	D_0	D_1	D_2
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

The implementation



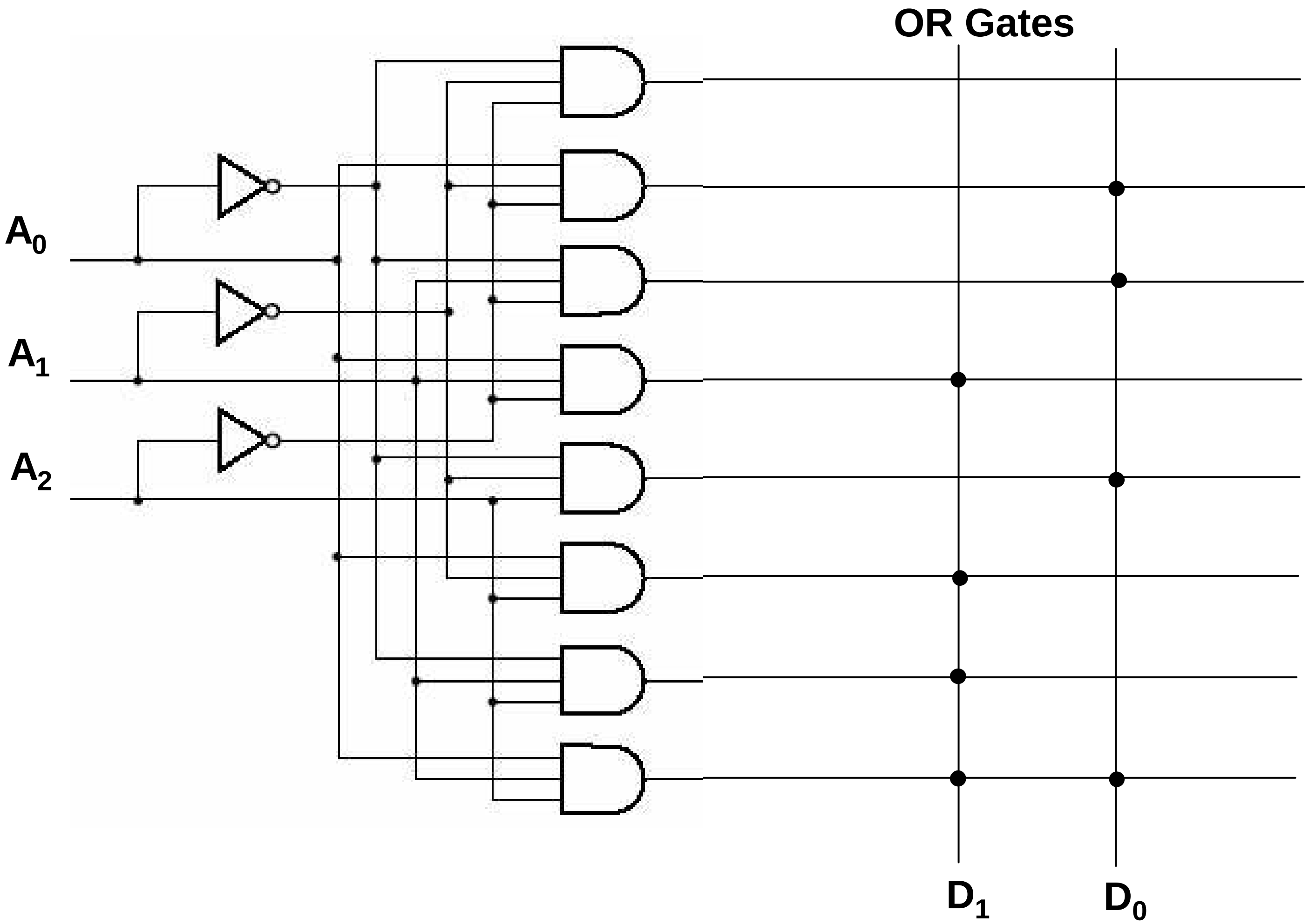
Exercise

- Build a ROM-based combinatorial circuit with
  - INPUT: 3 boolean variables
  - OUTPUT: the number of the 1s in the input

Solution: Truth Table

A_0	A_1	A_2	D_1	D_0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Solution: the implementation



Binary sum

Addend-1	0+	0+	1+	1+
Addend-2	0=	1=	0=	1+
Sum	0	1	1	0
Carry	0	0	0	1

It's just a 2-input, 2-output boolean function !

Called **half sum** since it ignores the carry-in

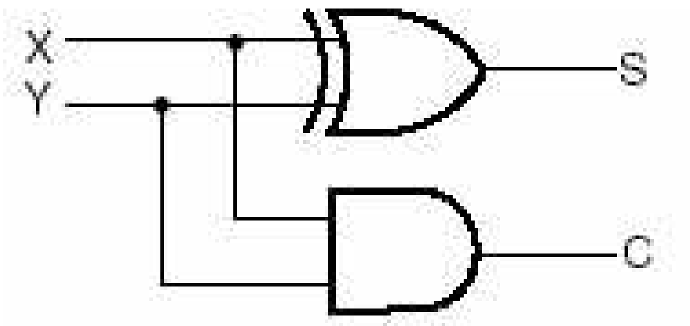
The half adder

- Sum two binary inputs without the carry-in

Truth table

X	Y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Logic Circuit



The complete addition

- Has to be able to deal with the carry-in

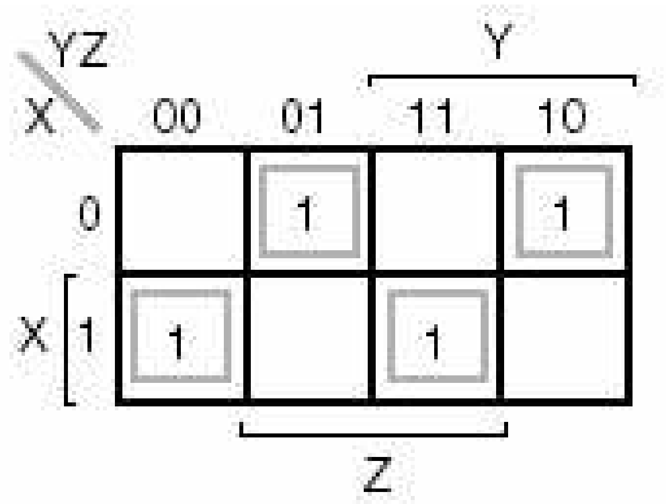
It's called **full adder**

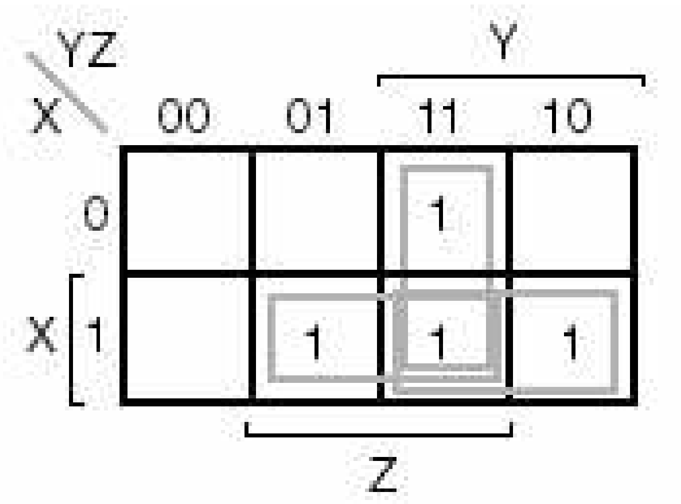
Z represents the carry-in

Truth table

X	Y	Z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Karnaugh's maps for full adder





$S = X'Y'Z+X'YZ'+XY'Z'+XYZ$

$= X \oplus Y \oplus Z$

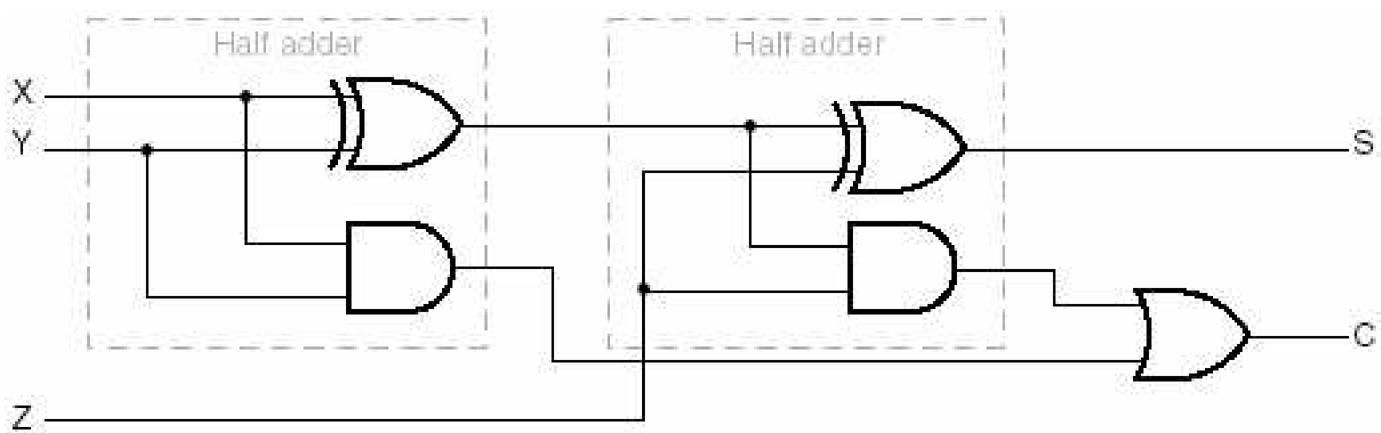
$C = XY + XZ + YZ$

$= XY + XY'Z + X'YZ$

$= XY + Z.(XY'+X'Y)$

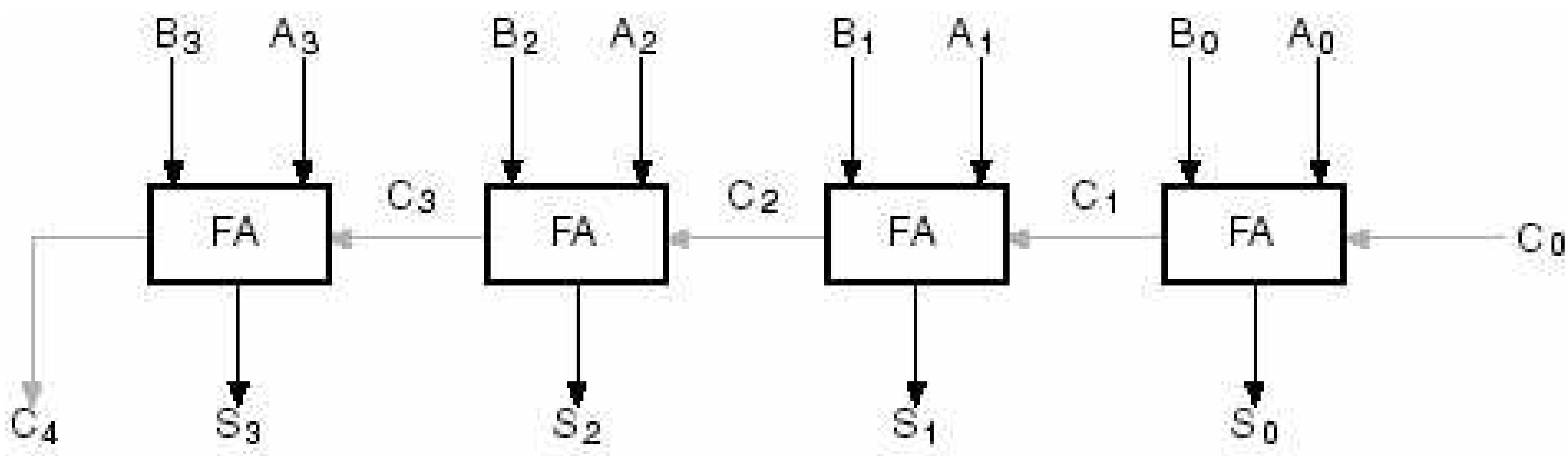
$= XY + Z.(X \oplus Y)$

The logic circuit of a full adder



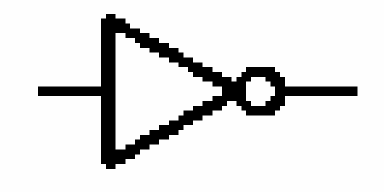
Binary adder

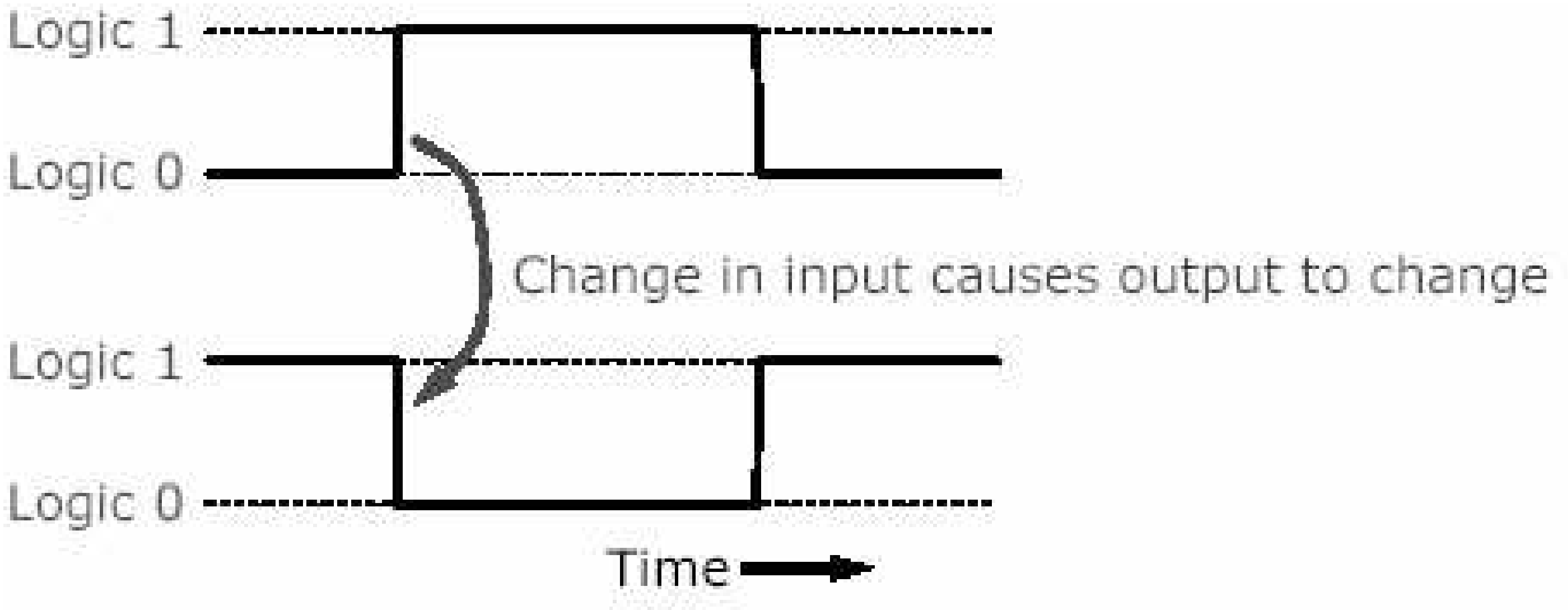
- Has to be able to deal with more bits
- An *n*-bit adder can be built chaining *n* full adders
- It's called **ripple-carry** adder



Ideal behaviour of circuits

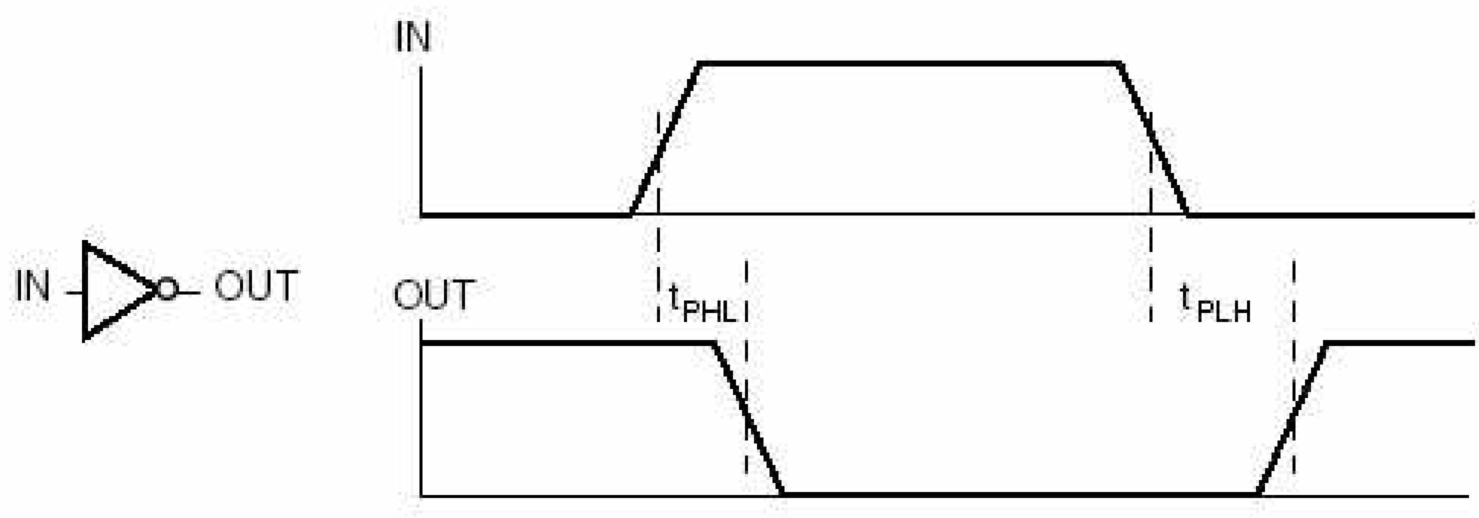
- Consider an inverter (NOT gate)





The real behaviour

- Propagation delay: time needed for a change in the input to affect the output (**gate delay**)
- Fall time: time taken for the signal to fall from high level to low level
- Rise time: time taken to rise from low to high



Carry Propagation

- Signals must propagate from inputs for output to be valid
- Carry and sum outputs of a single full-adder are valid *c* "gate-delays" after inputs are stable
- Value of *c* depends on the used technology
- In a binary adder of *n* bits the last carry is valid $c \cdot n$ "gate-delays" after inputs are stable
- For *n* large it may be unacceptable !

Solution

- Pre-compute all carry-ins:
  - **carry look-ahead** adder
- Write a general expression for a carry
  - When does an input carry propagates to the output?
  - When is a carry generated in the output?

X	Y	Z	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

General expression

- General expression for the (i+1)-th carry
  - $c_{i+1} = x_i y_i + c_i (x_i + y_i) = g_i + c_i p_i$
  - $g_i \rightarrow$ generate carry
  - $p_i \rightarrow$ propagate carry
  - Iterate the expression for c_{i+1}

General expression (2)

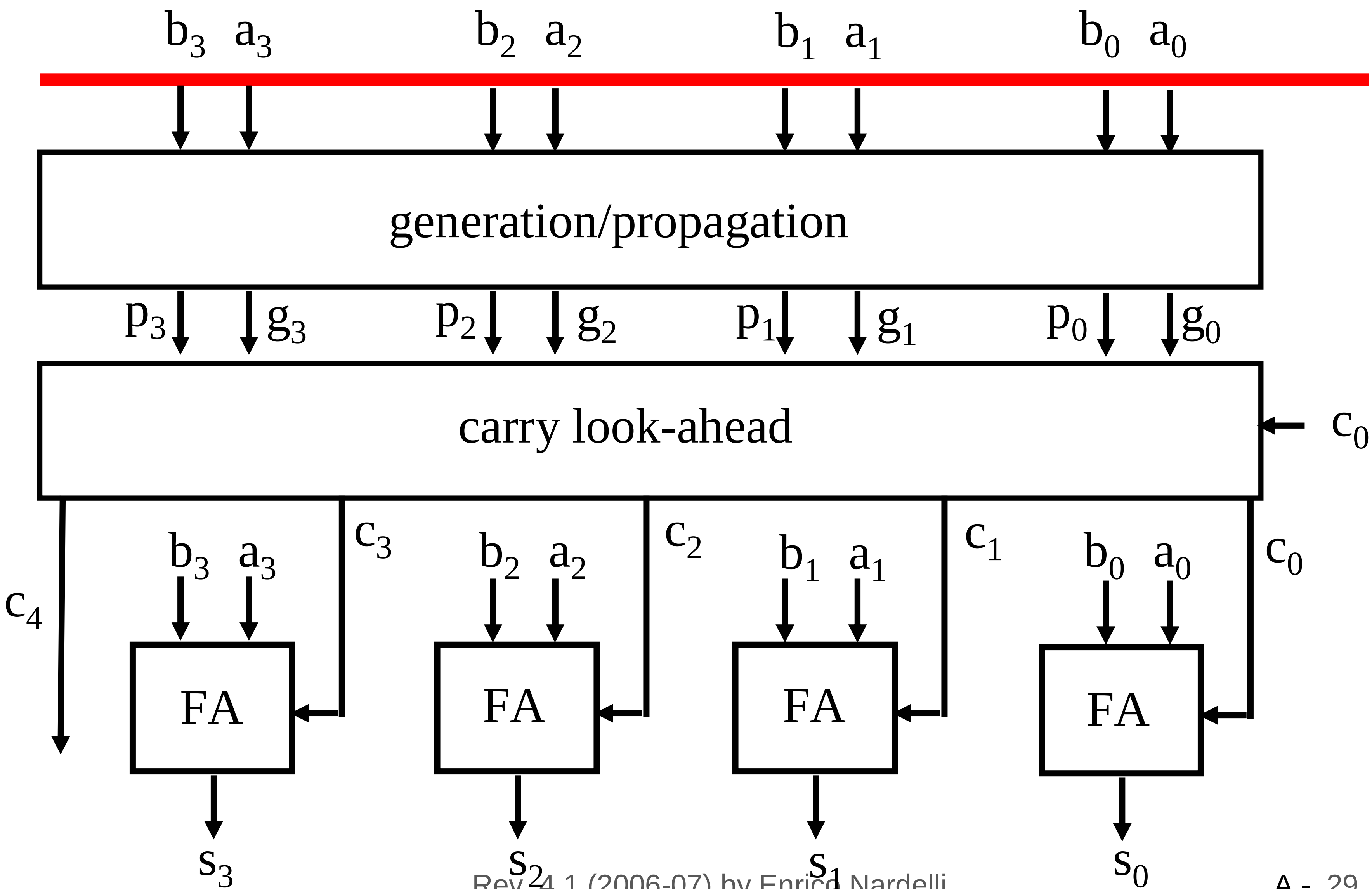
$$c_{i+1} = g_i + p_i(g_{i-1}+c_{i-1}p_{i-1}) = g_i+p_ig_{i-1}+p_ip_{i-1}c_{i-1} =$$
$$= g_i+p_ig_{i-1}+p_ip_{i-1}(g_{i-2}+c_{i-2}p_{i-2})$$
$$= g_i+p_ig_{i-1}+p_ip_{i-1}g_{i-2}+p_ip_{i-1}p_{i-2}c_{i-2}$$
$$= g_i+p_ig_{i-1}+p_ip_{i-1}g_{i-2}+p_ip_{i-1}p_{i-2}g_{i-3}+p_ip_{i-1}p_{i-2}p_{i-3}g_{i-4}+\ldots$$

- It could be developed until the least significant input bits
- Every c_i depends only on c_0, p_j, g_j $(j<i)$

Carry expressions for a 4-bit adder

- $c_1 = g_0 + p_0c_0$
- $c_2 = g_1 + p_1g_0 + p_1p_0c_0$
- $c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$
- $c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

Carry Look-Ahead: the architecture



A pratical problem

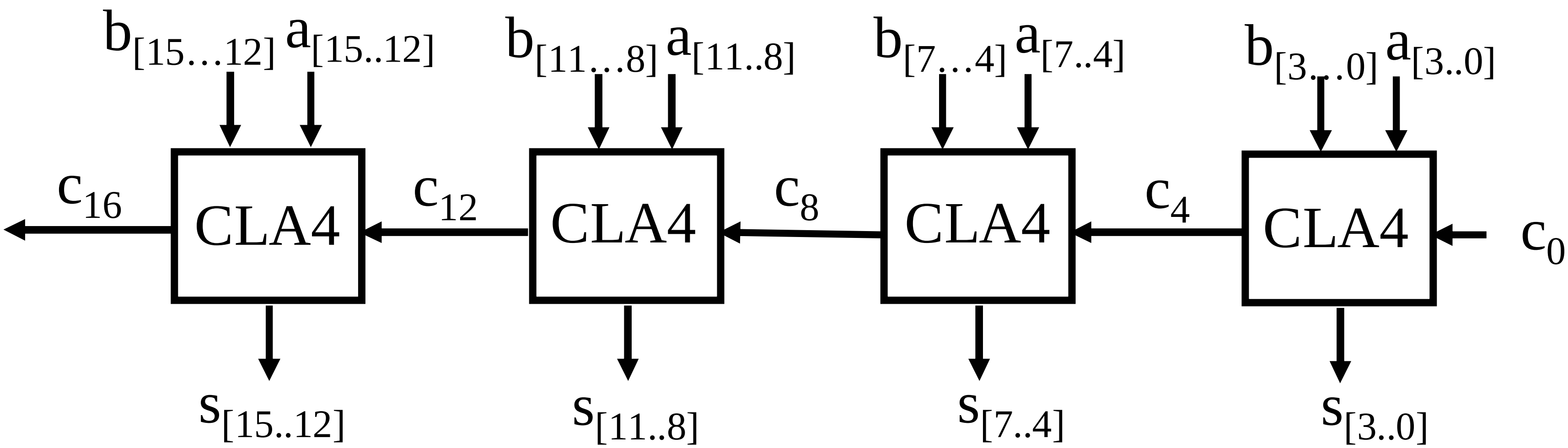
- $c_1 = g_0 + p_0c_0$
- $c_2 = g_1 + p_1g_0 + p_1p_0c_0$
- $c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$
- $c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

there is a limit due to circuit **fan-in**:
the maximum number of inputs

Practical solution for *n* bits

- Use carry look-ahead adders for just *m* consecutive bits (4-8 is typical)
- Each of these is a *stage*
- Use *n/m* stages connected by means of the ripple-carry technique
- The overall delay is now only $c \cdot n/m$ "gate delays"

A mixed solution

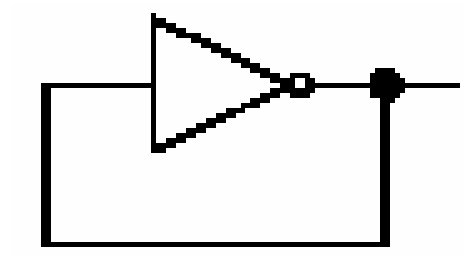


Sequential circuits

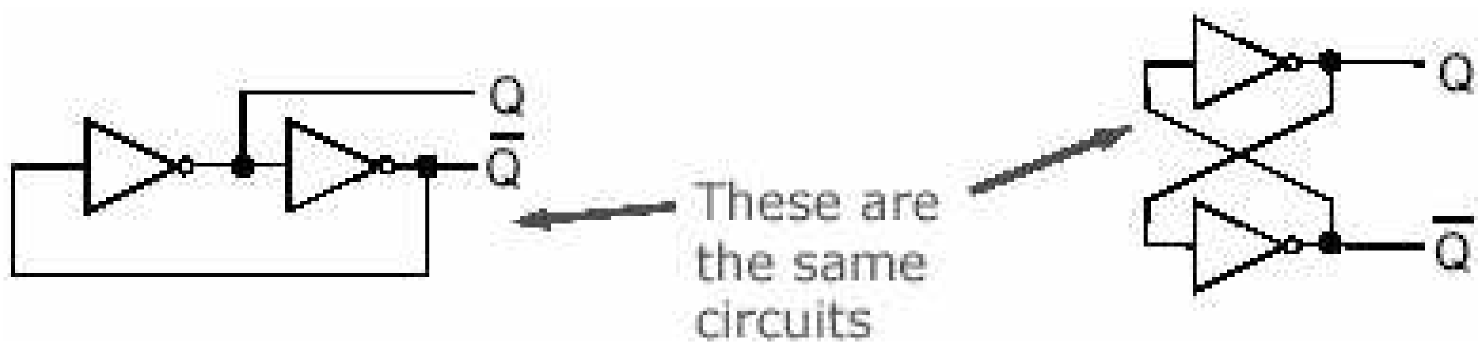
- More difficult to analyze since there is **feedback**: output is *fed back* to input
- Need to introduce a concept of state
  - **Current** state and **next** state
- **Asynchronous**: change of state of an element is fed into other elements without any coordination
- **Synchronous**: change of state of each element is fed into other elements only at a given instant, the same for all elements

Initial examples

- What does this circuit do ?



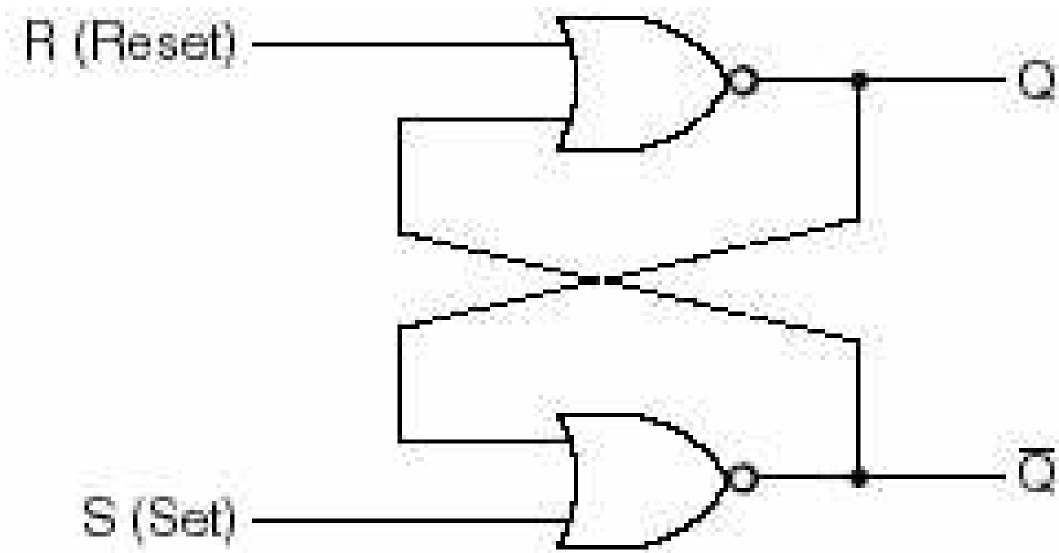
- What about this one ?



- Replace inverters with NOR gates

SR-Latch

- Analyze this circuit (write truth table)



Analysis of SR-Latch

- Two kinds of analysis
- COMBINATIONAL
  - Consider all possible configurations of S,R,Q and check their feasibility
- SEQUENTIAL
  - Consider all possible configurations of S,R,Q at a generic step *k* and check what happens for Q at step *k+1*

SR-Latch Truth Table: Combinational View

- 8 possible combinations (Q= NOT Q')

#	S	R	Q	Q'	*name*
0	0	0	0	1	Keep
1	0	0	1	0	Keep
2	0	1	0	1	Reset
5	1	0	1	0	Set
7	1	1	1	0	*Unfeasible combination*
3	0	1	1	0	*Unfeasible combination*
4	1	0	0	1	*Unfeasible combination*
6	1	1	0	1	*Unfeasible combination*

SR-Latch Truth Table: Sequential View

- Next state as a function of current state

#	S	R	Q(k)	Q(k+1)	Q′(k+1)	*name*
0	0	0	0	0	1	Keep (stable)
1	0	0	1	1	0	Keep (stable)
2	0	1	0	0	1	Reset (stable)
3	0	1	1	0	1	Reset (*transient*)
5	1	0	1	1	0	Set (stable)
4	1	0	0	1	0	Set (*transient*)
6	1	1	0	0	0	*Transient but unacceptable !*
7	1	1	1	0	0	*Transient but unacceptable!*

First reason to avoid S=R=1

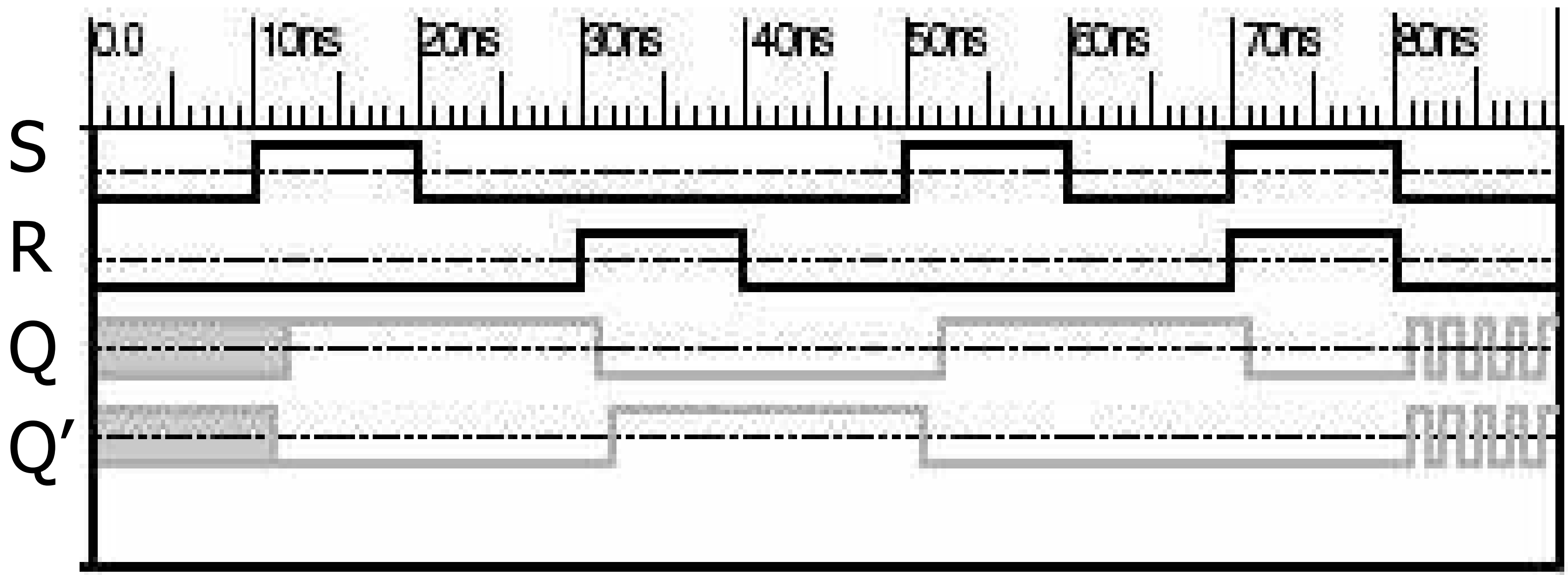
- When both inputs go from 1 to 0:
  - a **race** condition happens
- Both outputs are driven from 0 to 1
- Due to unpredictable physical differences one of the NOR gates may commute earlier from 0 to 1
- Then it will prevent the commutation of the other gate
- Conclusion: output value is unpredictable !

Second reason to avoid S=R=1

- When both inputs go from 1 to 0:
  - a **race** condition happens
- Both outputs are driven from 0 to 1
- Both the NOR gates commute from 0 to 1 almost at the same time
  - This drives both outputs from 1 to 0
  - Both gates are again forced to commute
  - This repeats again and again
- Conclusion: output values oscillate !

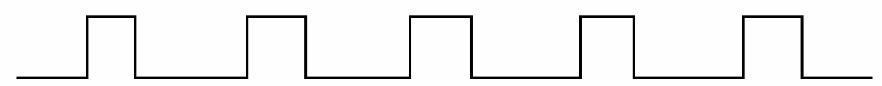
Temporal evolution of SR-latch

Time ->

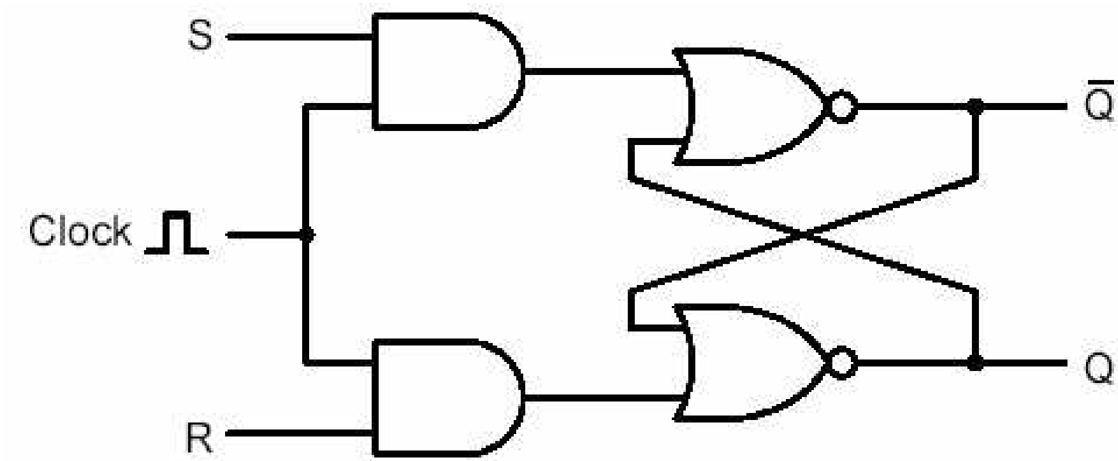


Adding a clock to SR-latch

- An additional input (the **clock**) is used to ensure the latch commutes only when required

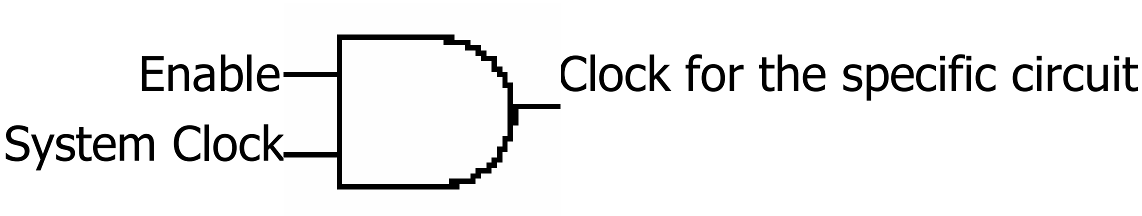


- The latch senses S and R only when Clock=1

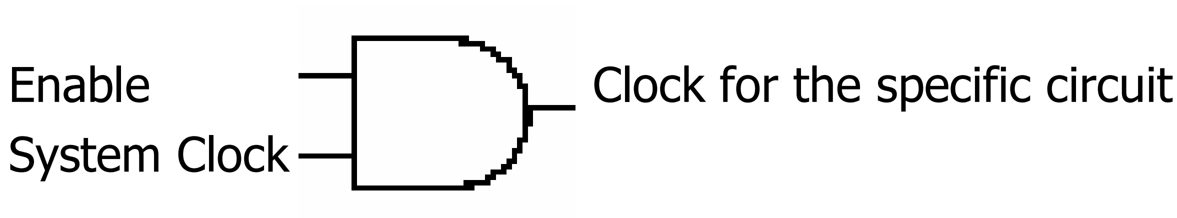


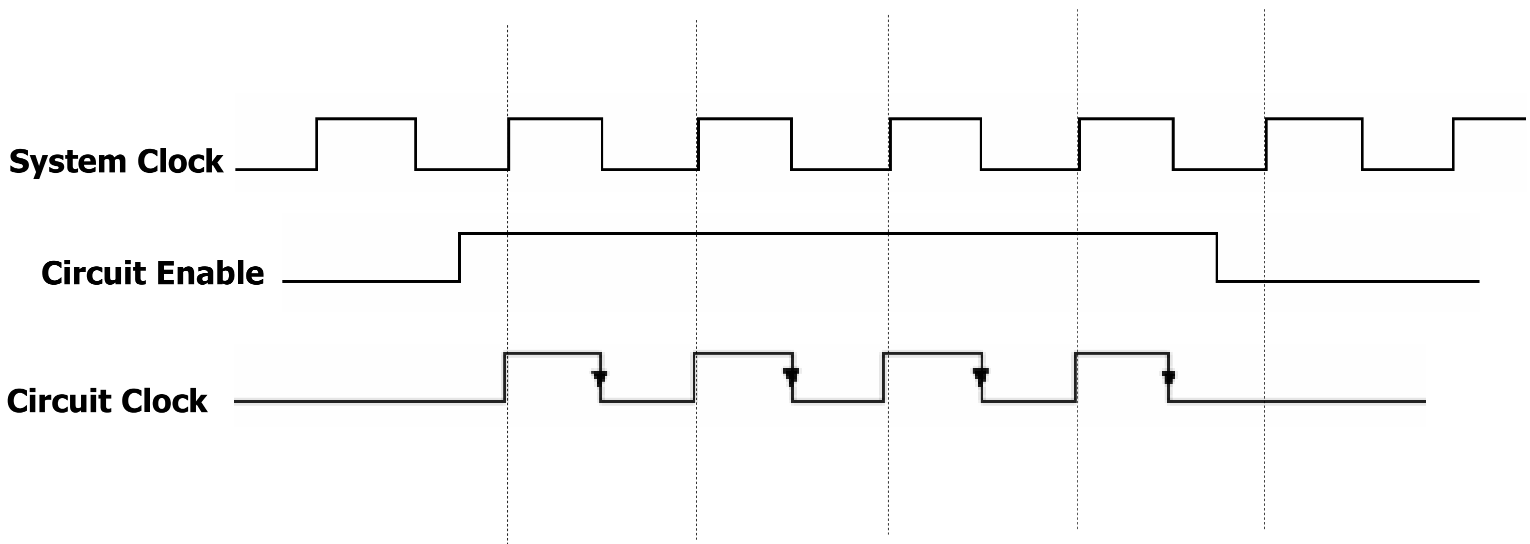
The role of the clock

- A clock ensures commutation is propagated from the input to the output only when required
- But the general system clock is running continuously: how can it be used to control a circuit only when needed?



Circuit clock from system clock





A more subtle problem

- Even if the circuit is clocked, inputs to the internal NOR gates receiving S and R may arrive with different delays
- In a commutation from (S=1,R=0) to (S=0,R=1) the SR-latch outputs may be (for some time) in the unacceptable state where both are 0

Example

- The NOR gate receiving the R:0-to-1 input may commute earlier than the other gate and now outputs of the SR-latch are in an unacceptable state

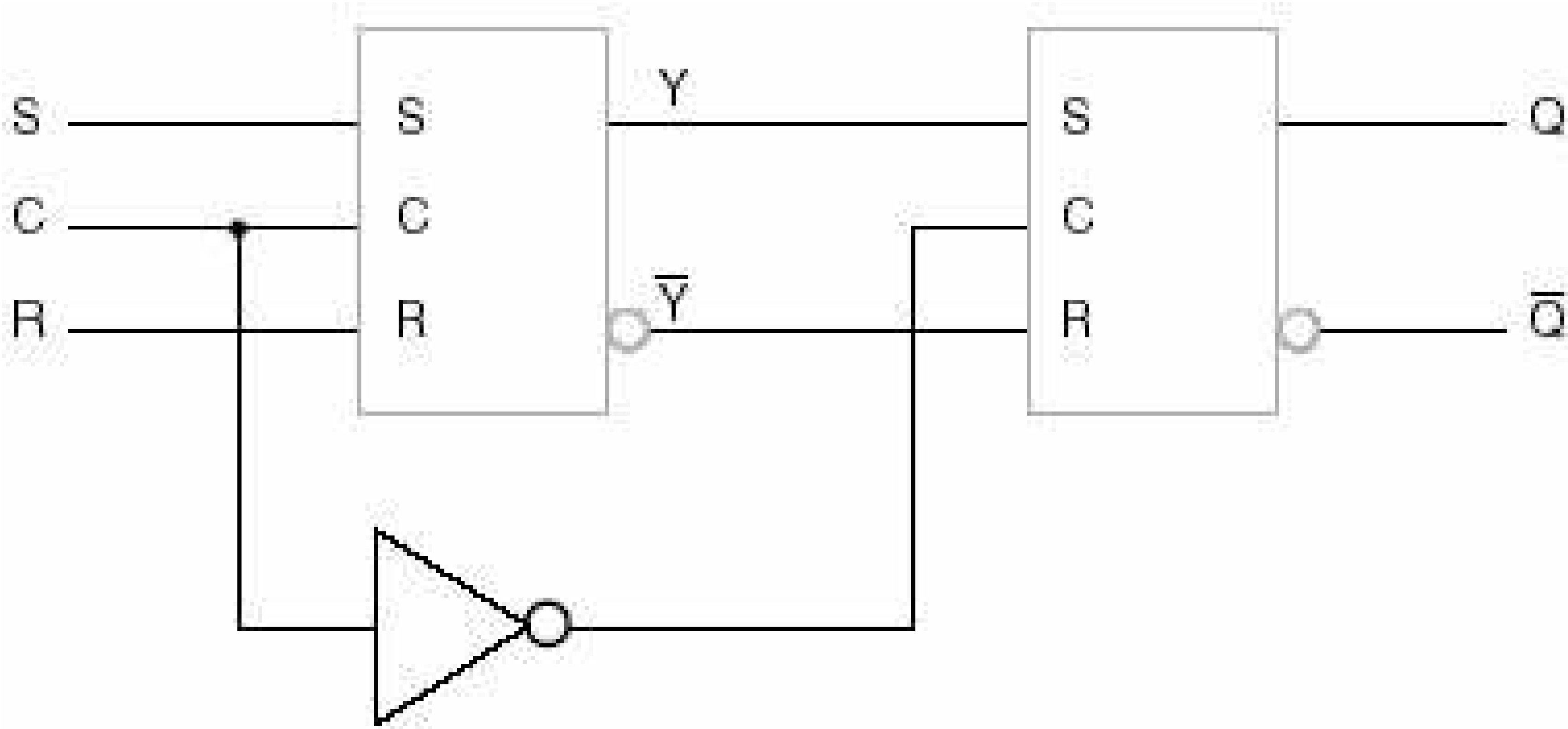
	Initial state	Input change	Transient state	Stable state
S	1	0	0	0
R	0	1	1	1
Q	1	1	0	0
Q’	0	0	0	1

- If Q and Q’ are in input to a further circuit, this receives wrong input values, hence its computed output may differ from the required one

The solution: a Flip-Flop circuit

- A 2-stage (*master* and *slave*) circuit
- First the master stage (connected to circuit's inputs only) changes its state (**flip**) when clock commutes to 1
- Then the slave stage (connected to circuit's outputs only) reads master's outputs after they have stabilized, when clock commutes to 0, and changes its state (**flop**)
- The next circuit will read slave's outputs in the next clock commutation to 1, when they have stabilized
- Outputs from the *i*-th circuit are read in input to the (*i+1*)-th circuit only after the transient unacceptable phase is ended, since adjacent stages are "active" only during different half periods of the clock
- In a chain of circuits this allows to control exactly when the (commuted) output of the *i*-th circuit acts on the input of the (*i+1*)-th circuit

SR flip-flop



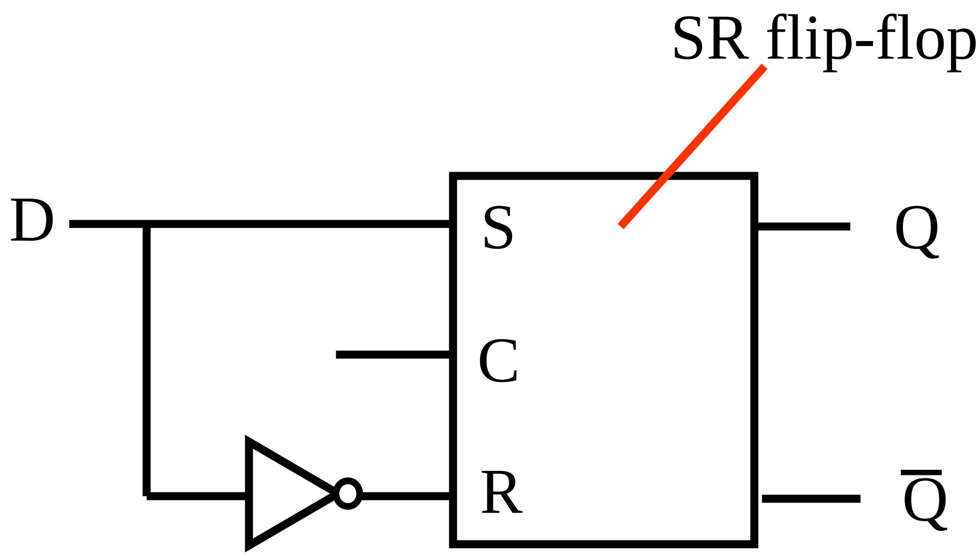
How SR flip-flop solve the problem

- Temporal evolution (both stages have a transient phase, but its effect on the next stage are hidden):

	In.	S↑	C↑	c1	C↓	c2	SR	C↑	Tr.	c1	C↓	Tr.	c2
S	0	1	1	1	1	1	0	0	0	0	0	0	0
R	0	0	0	0	0	0	1	1	1	1	1	1	1
C	0	0	1	1	0	0	0	1	1	1	0	0	0
C′	1	1	0	0	1	1	1	0	0	0	1	1	1
Y	0	0	0	1	1	1	1	1	0	0	0	0	0
Y′	1	1	1	0	0	0	0	0	0	1	1	0	1
Q	0	0	0	0	0	1	1	1	1	1	1	0	0
Q′	1	1	1	1	1	0	0	0	0	0	0	0	1

D flip-flop: a secure SR flip-flop

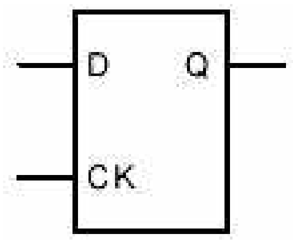
- Forcing R to always be NOT(S) the critical condition S=R=1 is avoided



Use of D flip-flop

- A D flip-flop is a memory cell, since it stores what is presented at its input

Symbol

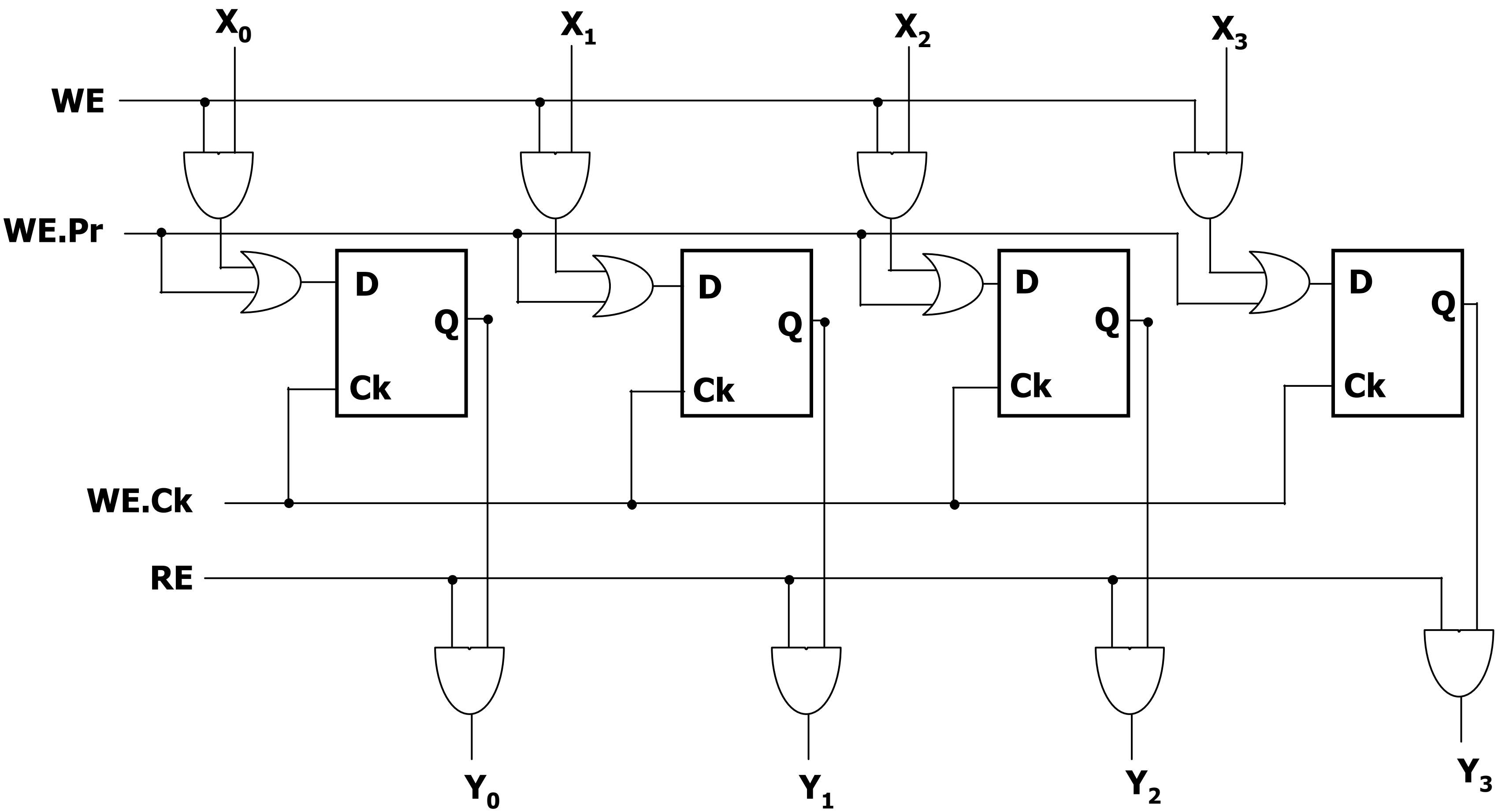


Truth table

D	Q_{n+1}
0	0
1	1

- Read-Enable (RE) and Write-Enable (WE) signals to store and read values
- Additional Preset (writes 1) and Clear (writes 0) signals to prepare the gate

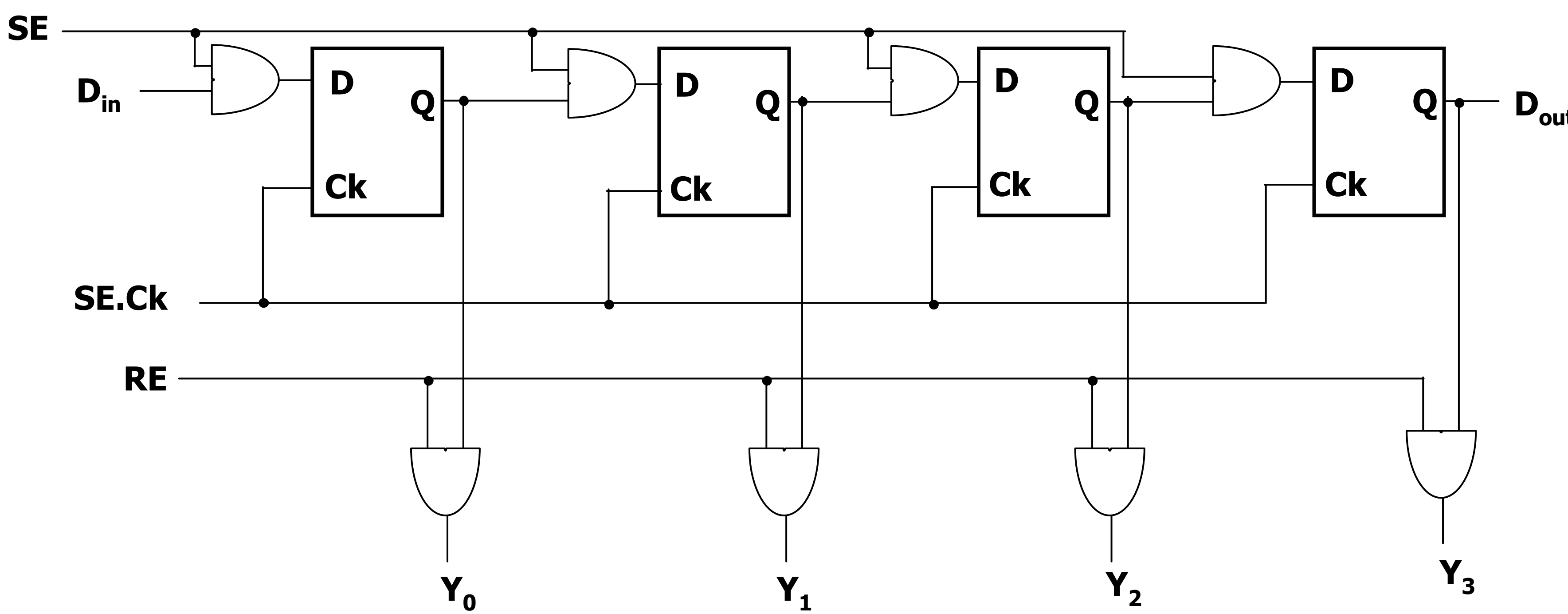
4 bit register



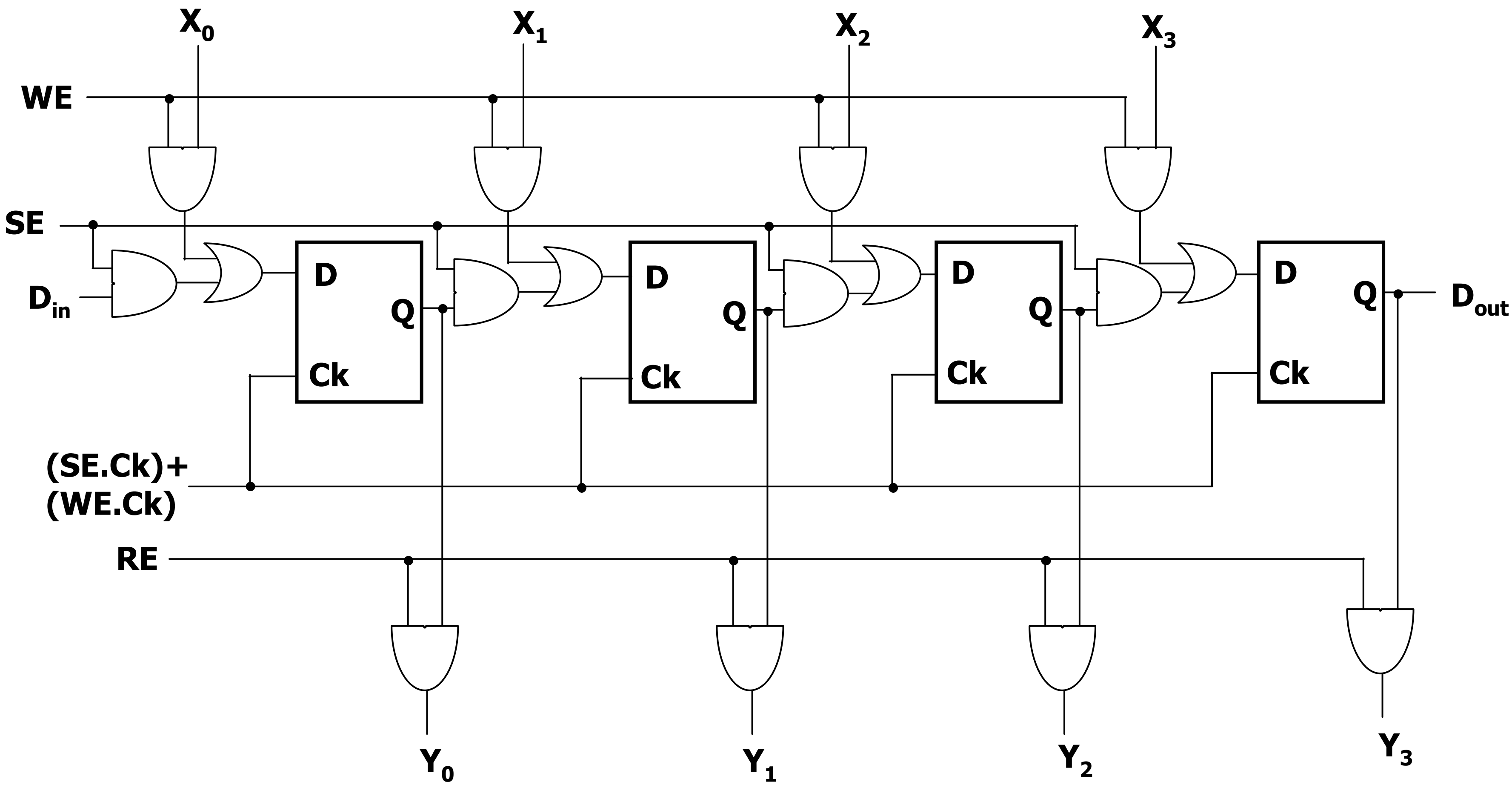
Use of D flip-flop (2)

- A D flip-flop is a delay unit, since it replicates at the output - one propagation delay later - what is presented at its input (delay flip-flop)
- A chain of *n* D flip-flops can be used to delay a bit value for *n* clock pulses

4 bit delay unit



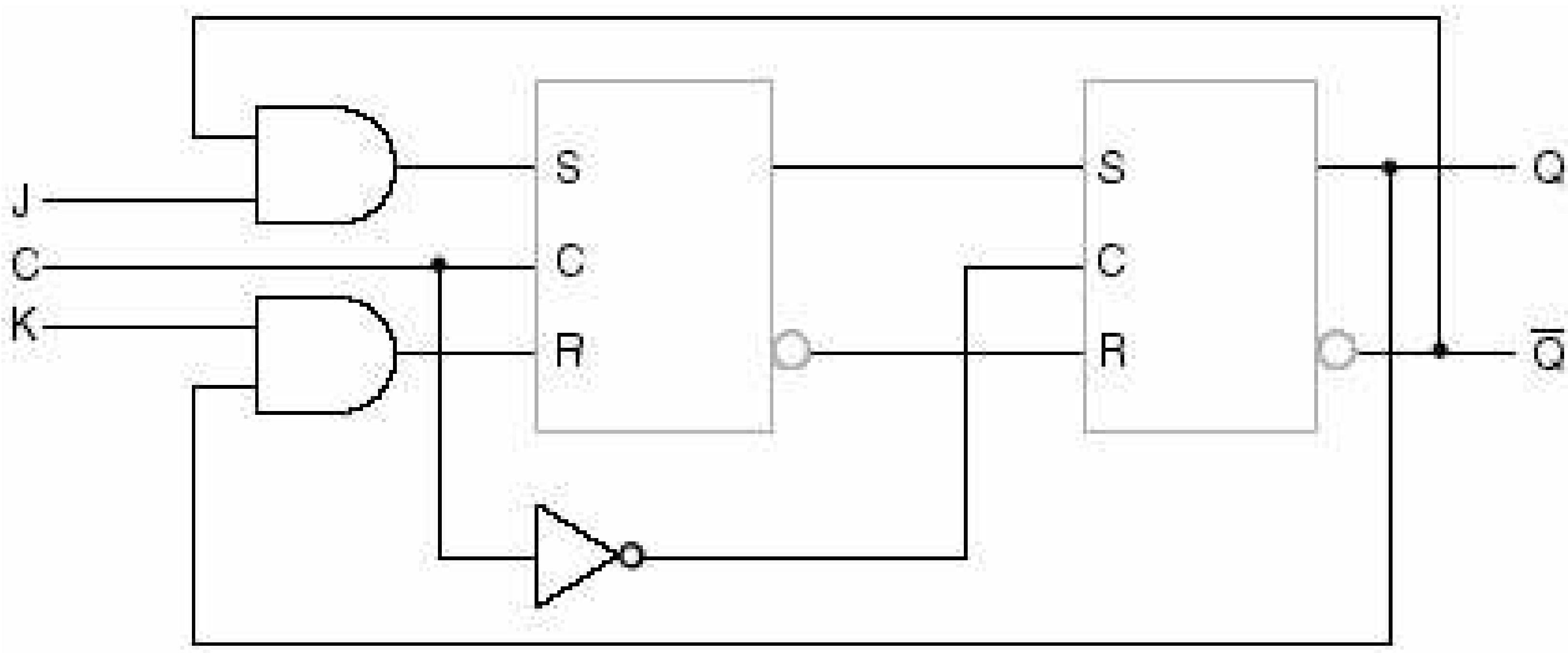
4 bit shift register



Register Control Signals

- WE (Write Enable): needed since many registers are attached to (i.e., receive data from) the same data bus
- SE (Shift Enable): allows a register output to drive next register input
- RE (Read Enable): needed since many registers are attached to (i.e., put data on) the same data bus

JK flip-flop: using also S=R=1



JK flip-flop: temporal evolution (1)

		J↑	C↑	C↓	J↓	C↑	C↓	K↑	C↑	C↓	J↑	C↑	C↓	C↑	C↓
J	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1
K	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Q	0	0	0	1	1	1	1	1	1	0	0	0	1	1	0
Q′	1	1	1	0	0	0	0	0	0	1	1	1	0	0	1
JQ′=S	0	1	1	0	0	0	0	0	0	0	1	1	0	0	1
KQ=R	0	0	0	0	0	0	0	1	1	0	0	0	1	1	0
Y	0	0	1	1	1	1	1	1	0	0	0	1	1	0	0
Y′	1	1	0	0	0	0	0	0	1	1	1	0	0	1	1

- Sequence of events

JK flip-flop: temporal evolution (2)

		J↓ K↓	C↑	C↓	K↑	C↑	C↓	J↑ K↓	C↑	C↓	K↑	C↑	C↓	C↑	C↓
J	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1
K	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1
Q	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1
Q’	1	1	1	1	1	1	1	1	1	0	0	0	1	1	0
JQ’=S	1	0	0	0	0	0	0	1	1	0	0	0	1	1	0
KQ=R	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
Y	0	0	0	0	0	0	0	0	1	1	1	0	0	1	1
Y’	1	1	1	1	1	1	1	1	0	0	0	1	1	0	0

- Sequence of events

Tabular description for JK-FF

- Input: J, K;   State: Q;   Output: Q

J	K	Q_n	Q_n
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	1
0	1	1	1
1	0	1	1
1	1	1	1

J	K	Q_n	Q_{n+1}
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	0

Transition Tables

- Synthetic description of flip-flop dynamics

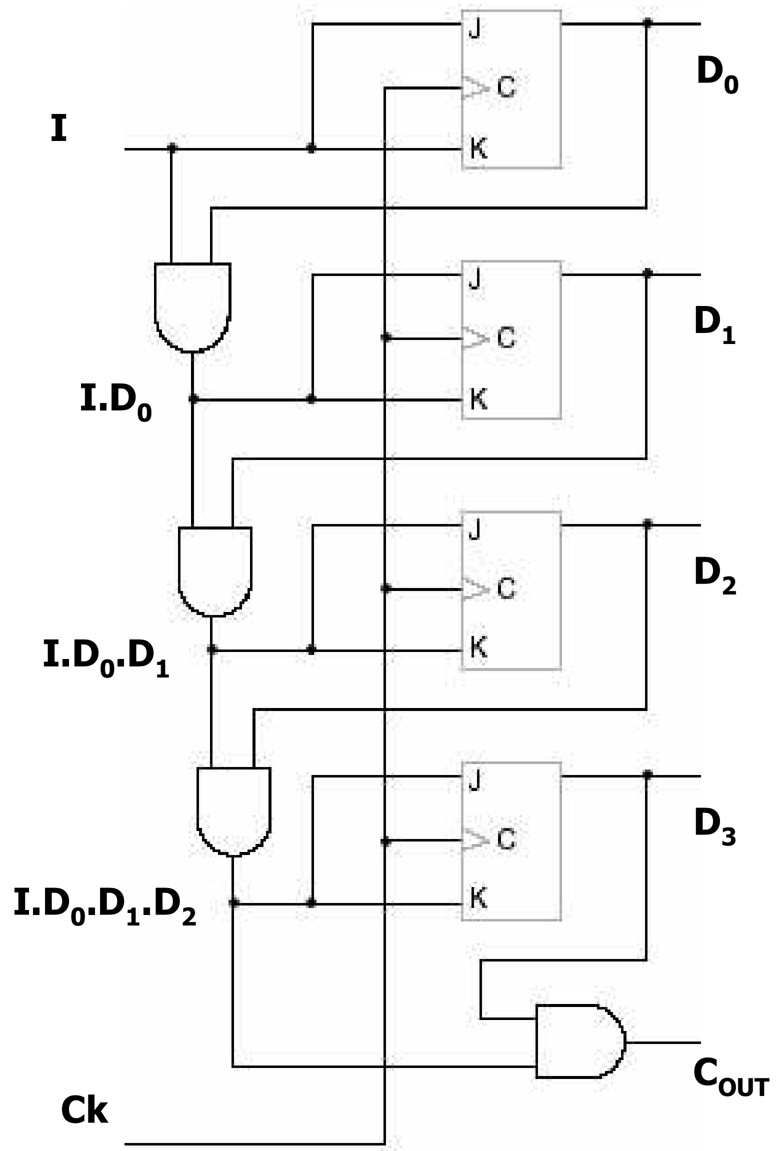
S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	---

D	Q_{n+1}
0	0
1	1

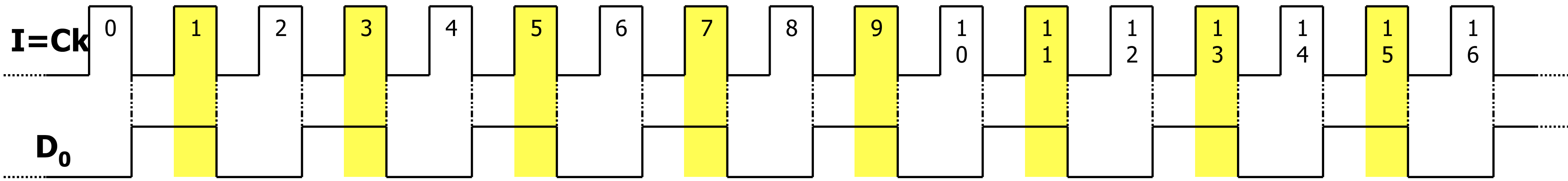
J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	Q'_n

Counters

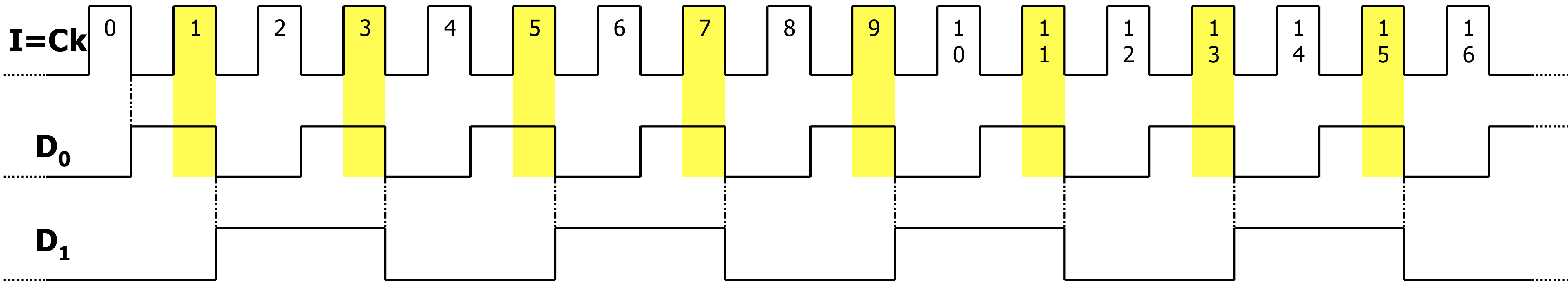
- IDEA: A single JK-FF with a periodic input commutes its output with twice the period of its input
- Use a chain of JK-FF each time doubling the period of the input
- A counter modulo 2^4 is shown



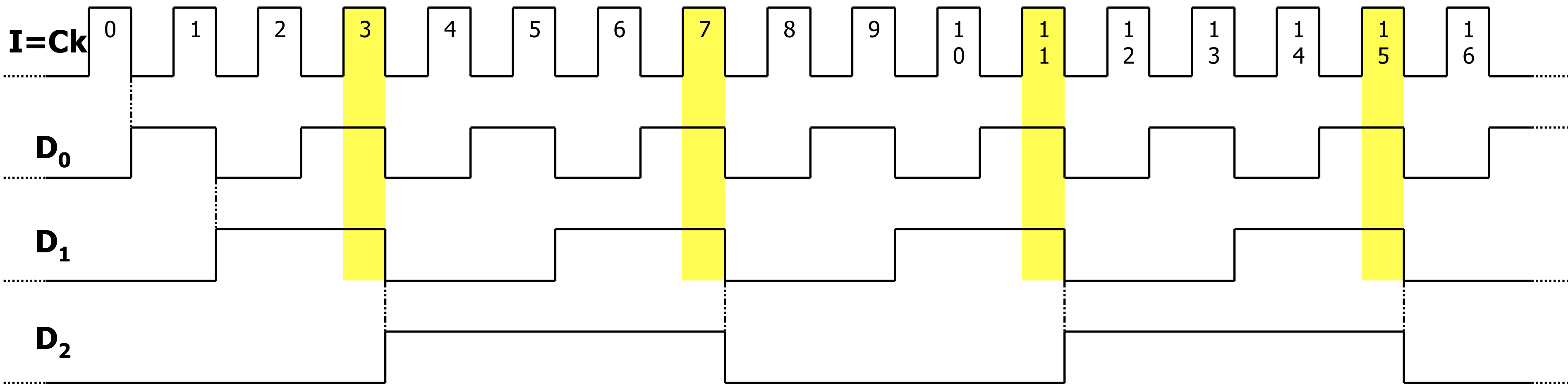
Temporal behaviour (1)



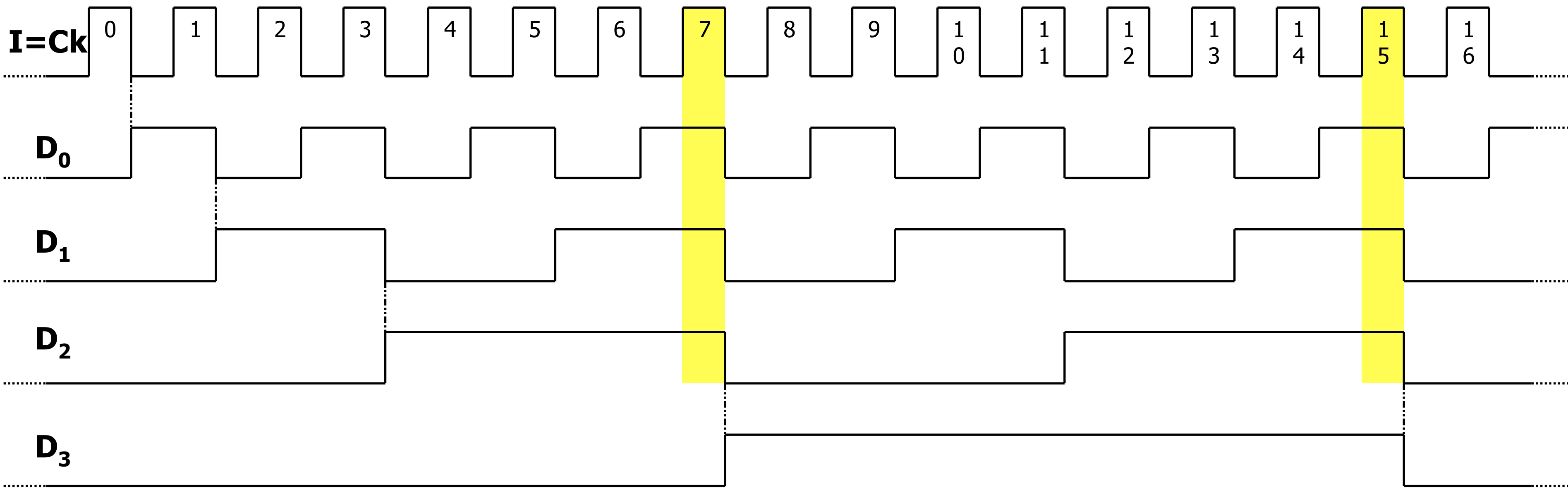
Temporal behaviour (2)



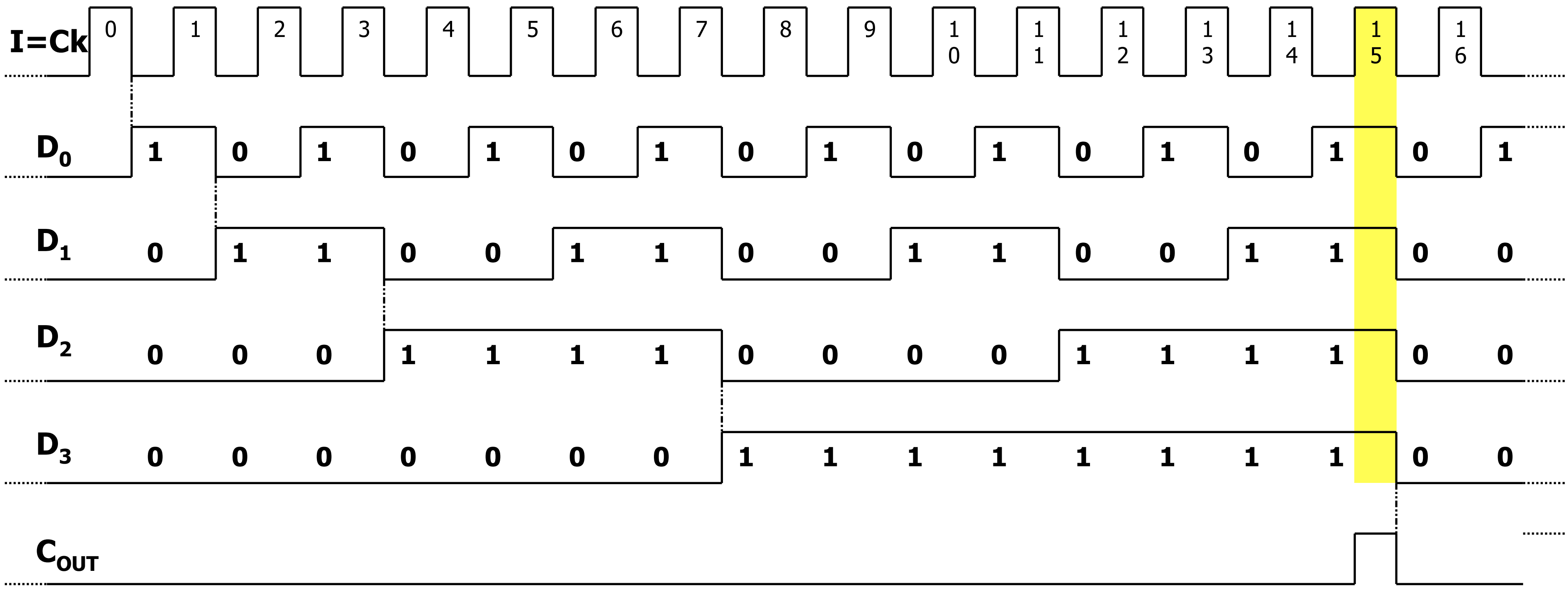
Temporal behaviour (3)



Temporal behaviour (4)

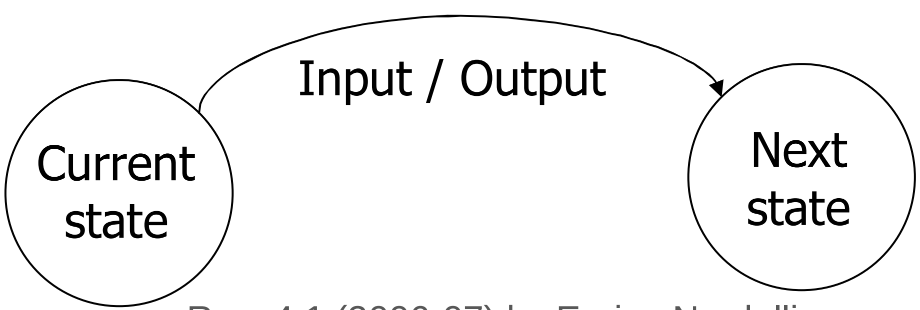


Temporal behaviour (5)



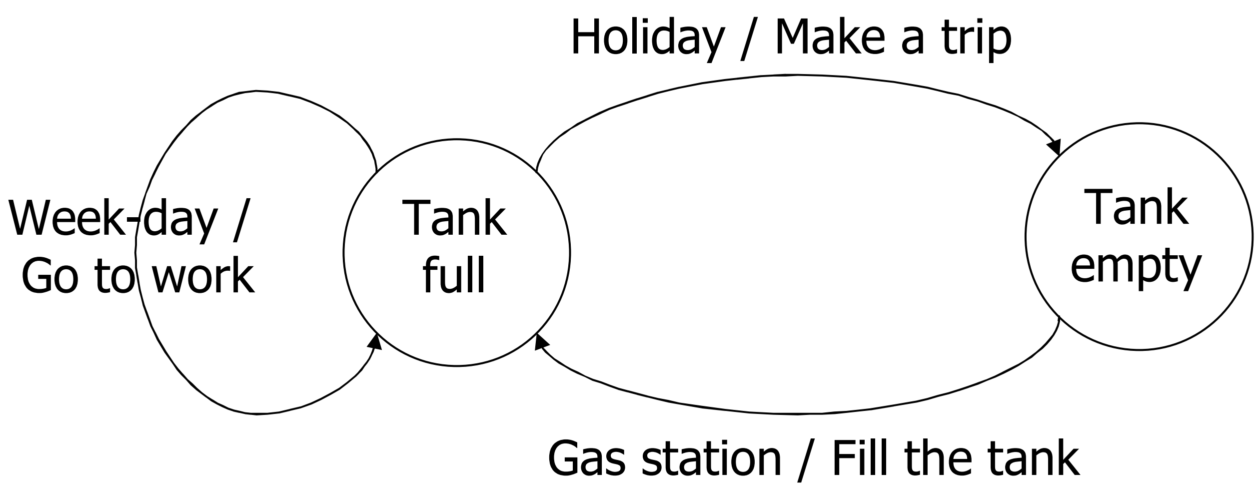
Finite State Machines (FSM)

- Called also Finite State Automata (FSA)
- Described by a table of **transitions** between **states** as a consequence of **inputs**
- If an input is true in a given state, a transition changes the state and may produce an **output**
- Graphical representation (states are circles, transition are arrows, input and output are arrow labels):



A very simple example of FSM

- When the tank of my car is *full*, if it is an *holiday* I *make a trip*, but if it is a *week-day* I *go to work* by bus. After the trip, the tank is *empty* and when I find a *gas station* I *fill* the tank



Tabular description for this FSM

- Next state as a function of current state and input

Current state	Input	Next state
Tank full	Holiday	Tank empty
Tank full	Week-day	Tank full
Tank empty	Gas station	Tank full

- Output as a function of current state and input

Current state	Input	Output
Tank full	Holiday	Make a trip
Tank full	Week-day	Go to work
Tank empty	Gas station	Fill the tank

Abstraction process

- FSM **describe** sequential networks (SN)
- SN **realizes** Finite State Machines

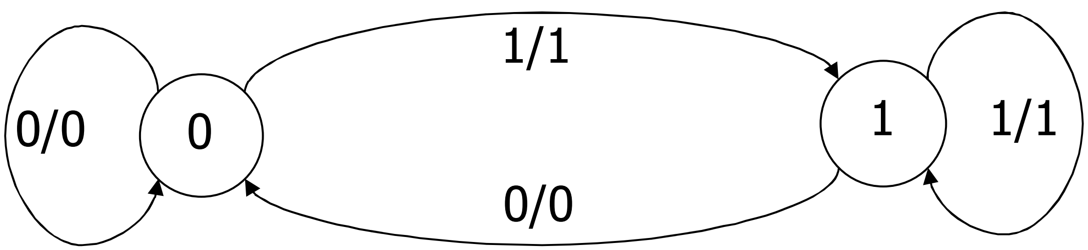
- The **analysis** of a SN allows to write the corresponding FSM
- From a FSM a SN is obtain through a **synthesis** process

- Similar to boolean functions and logical circuits
  - Boolean Functions (BF) **describe** logical circuits (LC)
  - LC **realize** Boolean Functions

  - The analisys of a LC produces a BF
  - LC are combinational networks (memoryless) synthesizing BF

FSA for D flip-flop

- Use Q as state descriptor (**state variable**)
- Use D as input
- Use Q as output
- Check for completeness



Its tabular description

- Output values as a function of input and current state values
- Next state values as a function of input and state value
  - D flip-flop

Output:

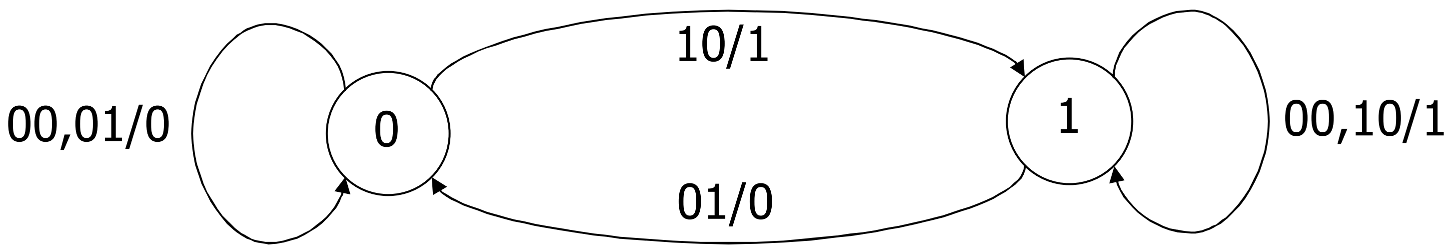
D	Q_n	Q_n
0	0	0
0	1	1
1	0	0
1	1	1

State:

D	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

FSA for SR flip-flop

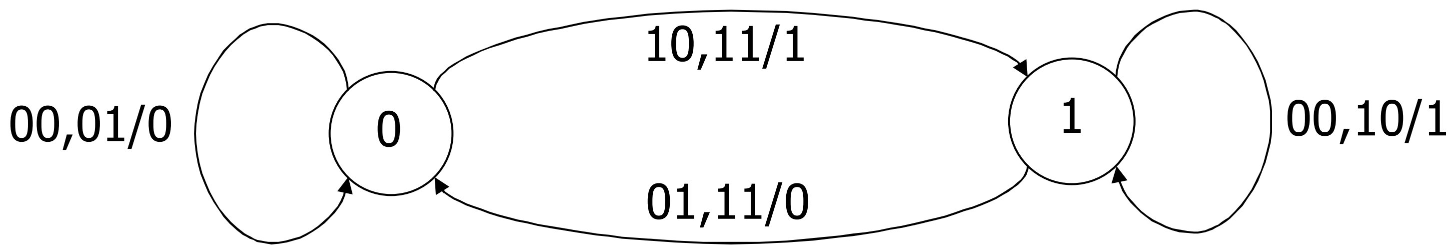
- Use Q as state variable
- Use S and R as input
- Use Q as output
- Transitions with multiple conditions



- Unacceptable input configurations are NOT represented

FSA for JK flip-flop

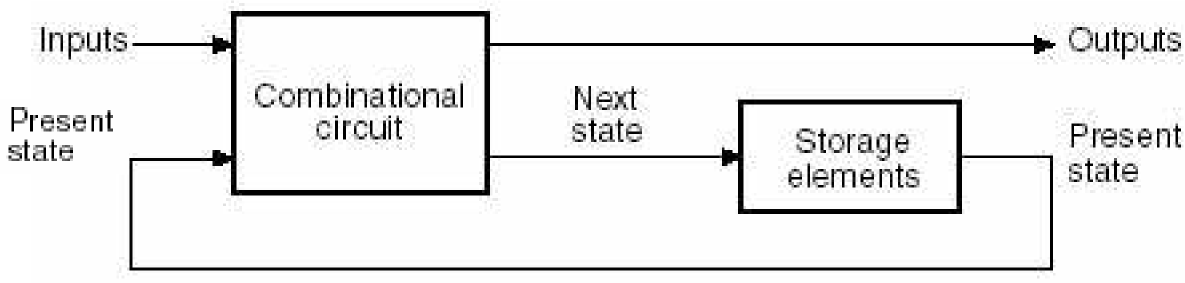
- Just add condition 11 to existing transitions
- Note stability and instability of states according to input values

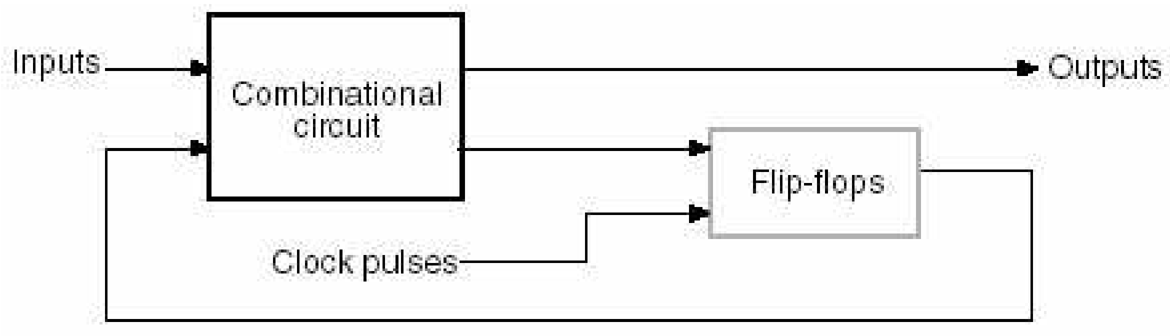


Synthesis of a SN from a FSA

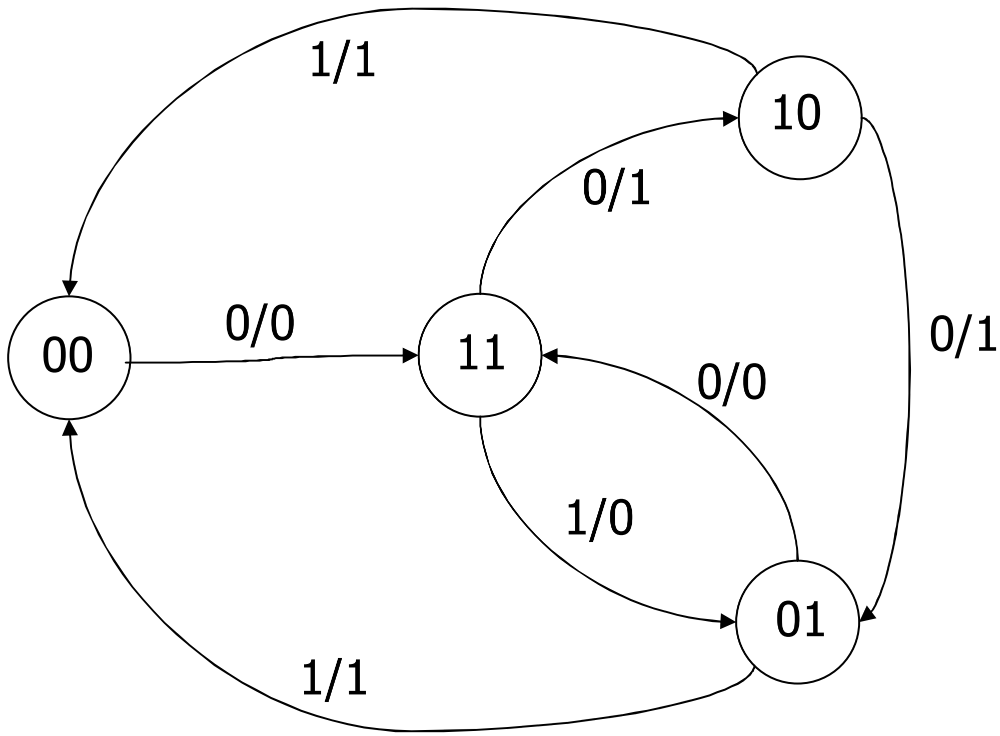
- Identify input, output and state variables
- Build (and minimize) truth tables for ouput variables as a function of input and state values
- Build (and minimize) transition tables for state variables as a function of input and state values
- Decide which FF to use to store state values
  - a D-FF is the simplest choice
  - to store 0 present 0 at the input
  - to store 1 present 1 at the input

Generic architecture of a SN

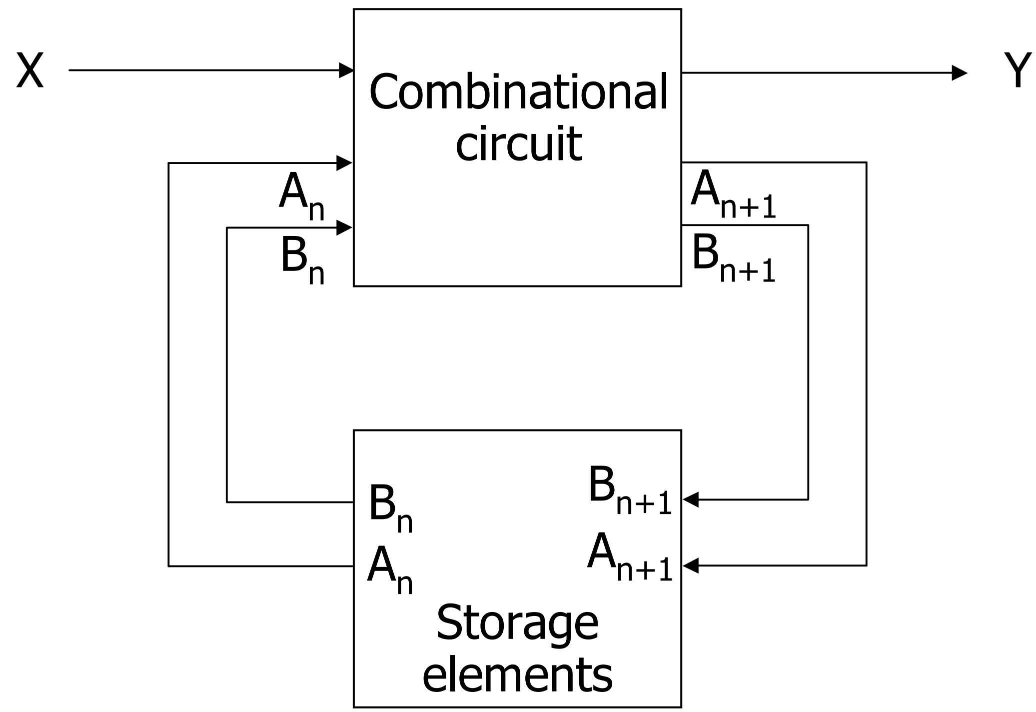




Example 1: a given FSA



Example 1: variables



Example 1: transition tables

- Transition table for output and state variables

A_n	B_n	X	Y	A_{n+1}	B_{n+1}
0	0	0	0	1	1
0	0	1	-	-	-
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	0	0	1

Example 1: minimization

State variables

A_{n+1}		X = 0	X = 1
AB	11	1	0
	01	1	0
	00	1	--
	10	0	0

B_{n+1}		X = 0	X = 1
AB	11	0	1
	01	1	0
	00	1	--
	10	1	0

Output

Y		X = 0	X = 1
AB	00	0	--
	01	0	1
	11	1	0
	10	1	1

- NOTE: Unspecified inputs cannot be used for minimization, otherwise a different FSM might be synthesized, i.e. an FSM with more transitions than in the initial specification !

Example 1: wrong minimization

- Using unspecified inputs for minimization means that we choose value 1 for those we take and 0 for the others …

State variables

A_{n+1}		X = 0	X = 1
AB	11	1	0
	01	1	0
	00	1	--
	10	0	0

B_{n+1}		X = 0	X = 1
AB	11	0	1
	01	1	0
	00	1	--
	10	1	0

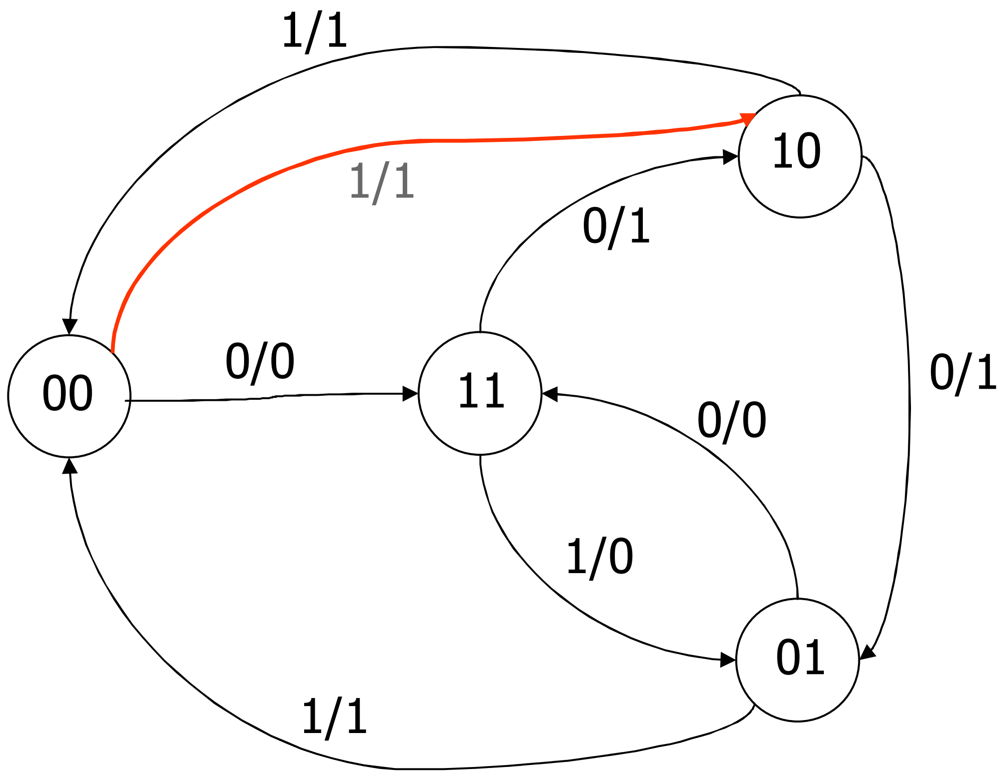
Output

Y		X = 0	X = 1
AB	00	0	--
	01	0	1
	11	1	0
	10	1	1

… hence we would implement …

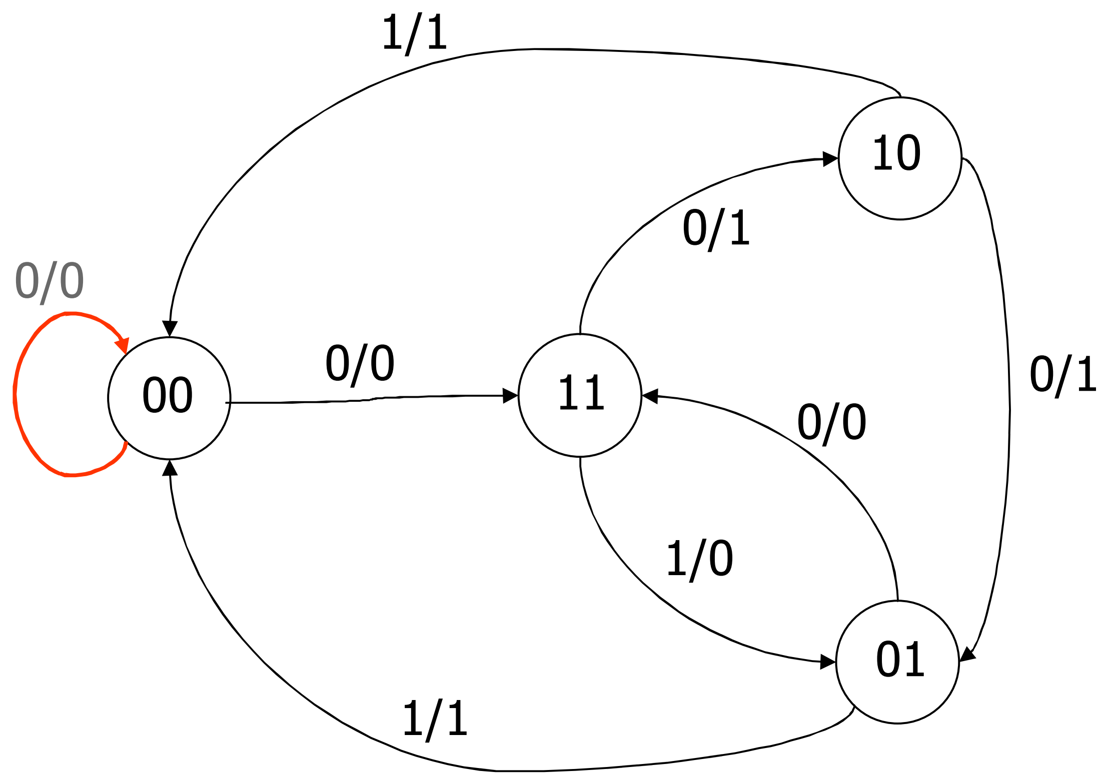
Example 1: the corresponding FSA

- …this FSA which has a different behavior from the original correct one!

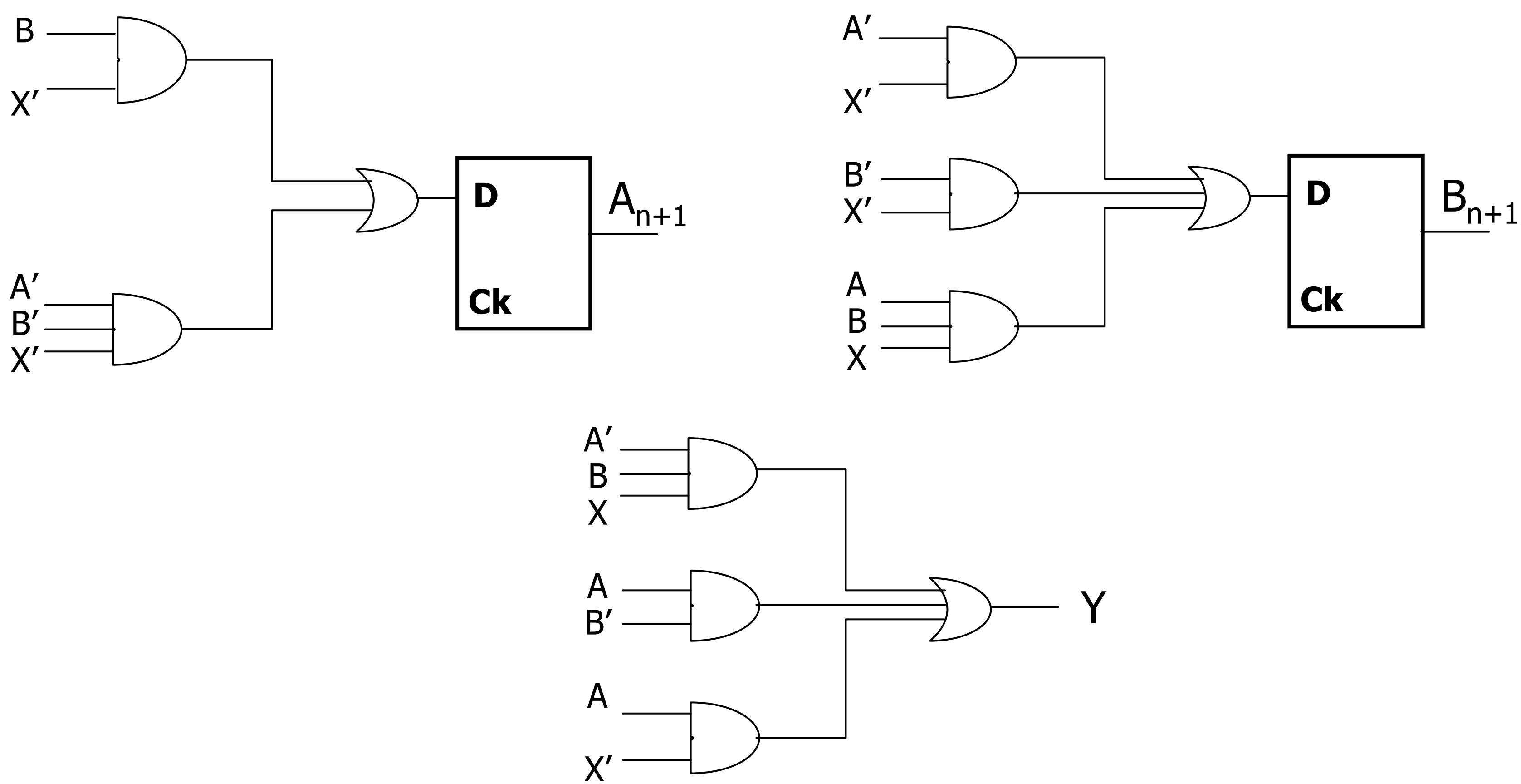


Example 1: one more comment...

- By not using unspecified inputs during minimization we are synthesizing this FSA, but this is an acceptable completion of the incompletely specified initial FSA!



Example 1: circuits



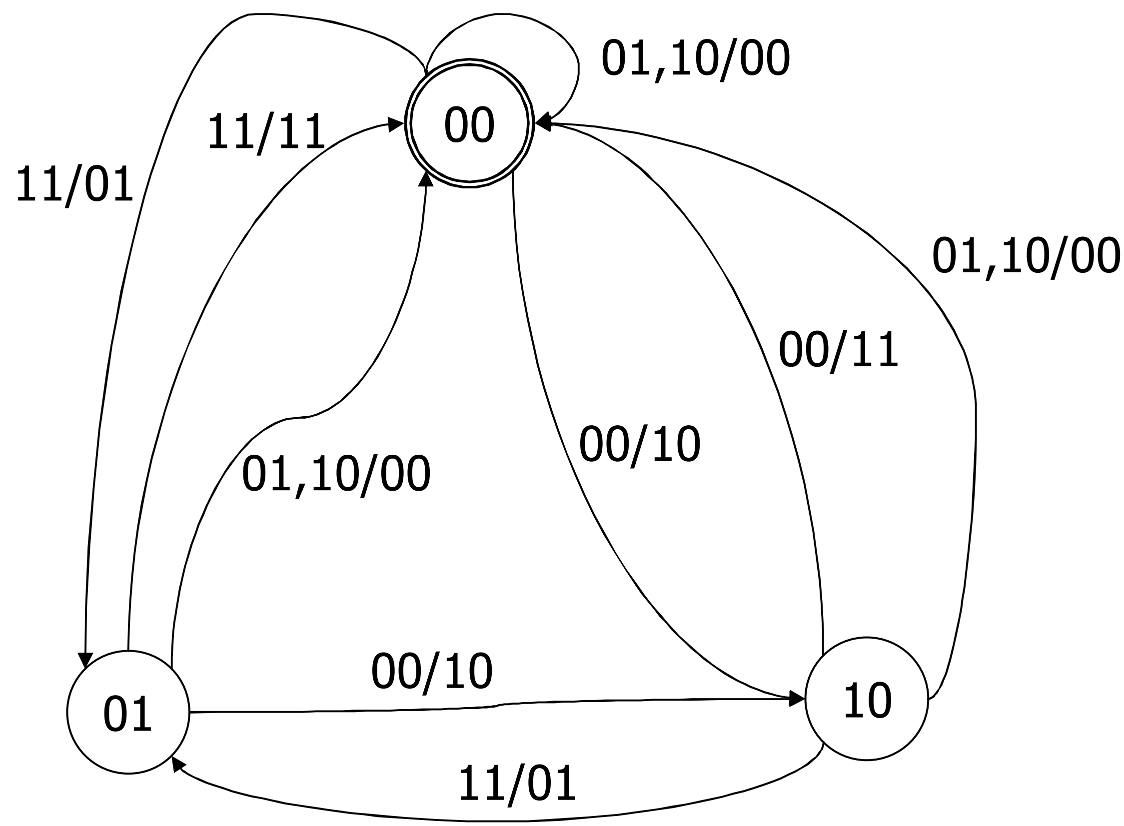
Example 2: specification

- Two input values are presented together
- Recognize with output 10 and 01, respectively, when a couple 00 or a couple 11 is presented
- Recognize with output 11 when two consecutive couples of identical values are presented
- Example:

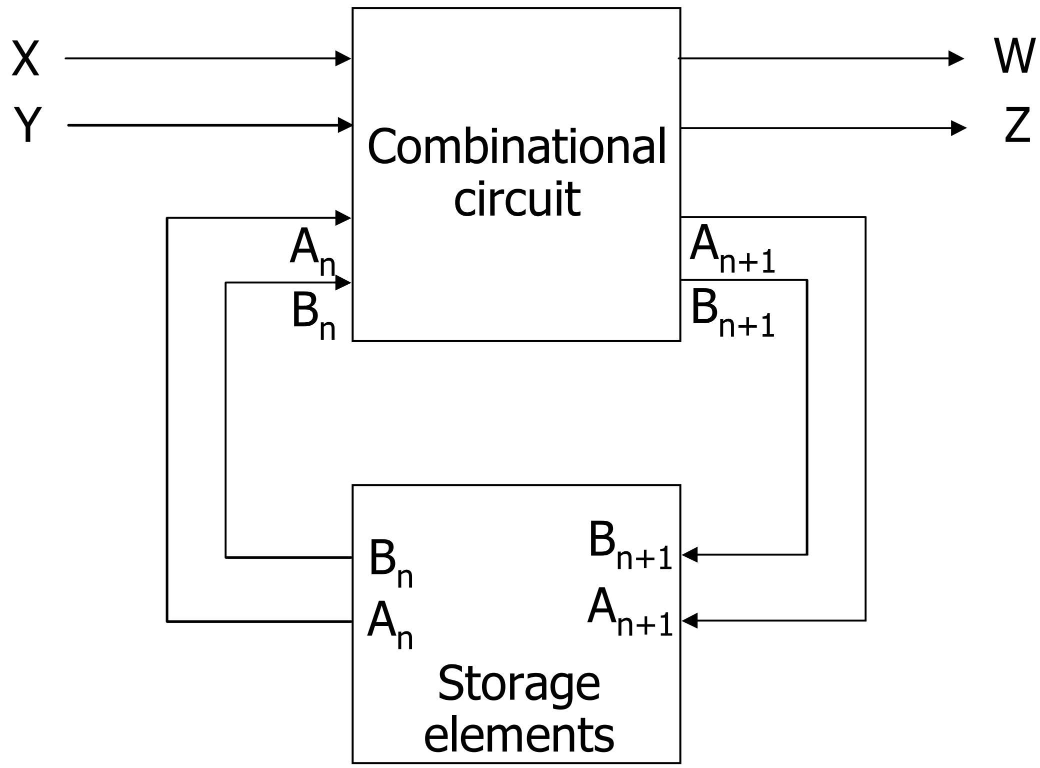
INPUT	01	01	00	00	00	11	11	10	11	11	11	11
OUTPUT	00	00	10	11	10	01	11	00	01	11	01	11

Example 2: corresponding FSA

- Show also the initial state (double circle)



Example 2: variables



Example 2: transition tables

A_n	B_n	X	Y	W	Z	A_{n+1}	B_{n+1}
0	0	0	0	1	0	1	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	1
0	1	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	1	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	0	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	1
1	1	0	0	--	--	--	--
1	1	0	1	--	--	--	--
1	1	1	0	--	--	--	--
1	1	1	1	--	--	--	--

Example 2: circuits